ENCICLOPEDIA PRACTICA DE LA

INFORMATICA APLICADA

23

El lenguaje C, próximo a la máquina

Aula de Informática

SINCLAIR AMSTRAD COMMODORE IBM



EDICIONES SIGLO CULTURAL

ENCICLOPEDIA PRACTICA DE LA

INFORMATICA APLICADA

23

El lenguaje C, próximo a la máquina

EDICIONES SIGLO CULTURAL

Una publicación de

EDICIONES SIGLO CULTURAL, S.A.

Director-editor:
RICARDO ESPAÑOL CRESPO.

Gerente:
ANTONIO G. CUERPO.

Directora de producción:
MARIA LUISA SUAREZ PEREZ.

Directores de la colección:
MANUEL ALFONSECA, Doctor Ingeniero de Telecomunicación y Licenciado en Informática
JOSE ARTECHE, Ingeniero de Telecomunicación

Diseño y maquetación:
BRAVO-LOFISH.

Dibujos:
JOSE OCHOA Y ANTONIO PERERA.

Tomo XXIII. **El lenguaje C, próximo a la máquina**
AULA DE INFORMATICA APLICADA (AIA)
ENRIQUE SERRANO, Ingeniero de Telecomunicaciones

Ediciones Siglo Cultural, S.A.

Dirección, redacción y administración:
Pedro Teixeira, 8, 2.ª planta (Ed. Iberia Mart I). Teléf. 810 52 13. 28020 Madrid

Publicidad:
Gofar Publicidad, S.A. San Benito de Castro, 12 bis. 28028 Madrid.

Distribución en España:
COEDIS, S.A. Valencia, 245. Teléf. 215 70 97. 08007 Barcelona.
Delegación en Madrid: Serrano, 165. Teléf. 411 11 48.

Distribución en Ecuador: Muñoz Hnos.

Distribución en Perú: DISELPESA.

Distribución en Chile: Alfa Ltda.

Importador exclusivo Cono Sur:
CADE, S.R.L. Pasaje Sud América. 1532. Teléf.: 21 24 64.
Buenos Aires - 1.290. Argentina.

ISBN del tomo: 84-7688-082-0.
ISBN de la obra: 84-7688-018-9.

Fotocomposición:
ARTECOMP, S.A. Albarracín, 50. 28037 Madrid.

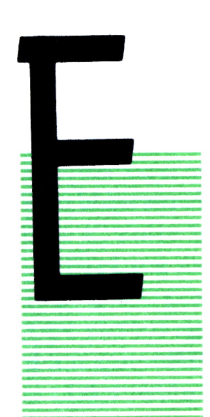
Imprime:
MATEU CROMO. Pinto (Madrid).

Depósito legal: M. 9.242-1987
Printed in Spain - Impreso en España.

Suscripciones y números atrasados:
Ediciones Siglo Cultural, S.A.
Pedro Teixeira, 8, 2.ª planta (Ed. Iberia Mart I). Teléf. 810 52 13. 28020 Madrid
Abril, 1987.
P.V.P. Canarias: 365,-

INDICE

HISTORIA DEL LENGUAJE C 1



EL lenguaje de programación C fue desarrollado por Dennis Ritchie, de los Laboratorios Bell, en la década de los setenta cuando trabajaba, junto con Ken Thompson, en el diseño del sistema operativo UNIX.

El UNIX surgió por la necesidad de disponer de un sistema operativo potente que pudiese funcionar en pequeños ordenadores de propósito general. Esto dio lugar a que Ken Thompson desarrollara una primera versión de un pequeño sistema operativo al que denominó UNIX, escrito en lenguaje ensamblador, siendo revisado posteriormente esta primera versión en un nuevo lenguaje de programación llamado «lenguaje B» desarrollado a su vez por Ken Thompson.

El lenguaje B fue ampliado por Dennis Ritchie, el cual obtuvo una nueva versión, a la que llamó «lenguaje C».

El momento actual del hardware, con el desarrollo de microordenadores de 16 bits que poseen la misma potencia que los miniordenadores de hace unos años, ha favorecido el uso del sistema operativo UNIX y el lenguaje C.

El lenguaje C se está transformando a pasos gigantescos en una de las bases de programación más importantes y populares.



CARACTERISTICAS FUNDAMENTALES DEL C

El C es hoy día el lenguaje por excelencia de los miniordenadores que trabajan bajo el sistema operativo UNIX.

Pero no solamente se trabaja con C en miniordenadores y grandes sistemas. Actualmente se dispone en el mercado de compiladores del lenguaje C para ordenadores personales.

C es un lenguaje de programación de propósito general que produce «programas compactos» y «eficientes».



Fig. 1. El sistema operativo Unix está escrito en C.

El diseño de programas en C puede llevarse a cabo bajo técnicas de programación estructurada (diseño «top-down»)

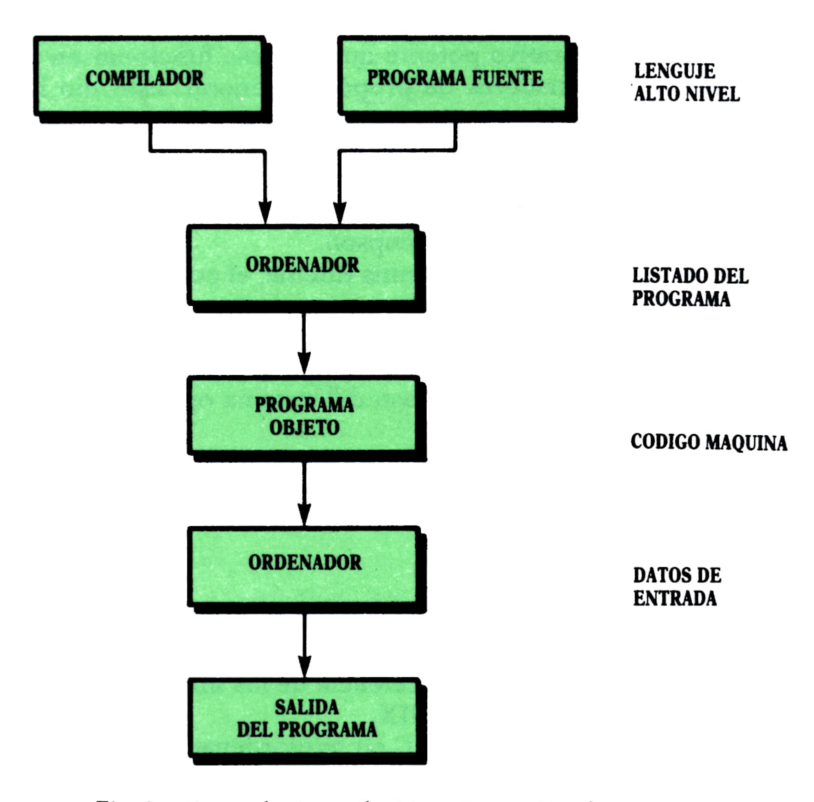


Fig. 2. Fases de Compilación y Ejecución de un programa.

incorporando estructuras de control básicas como DO, UNTIL, WHILE, SWITCH e IF-THEN-ELSE. Los aficionados a la programación reconocerán este tipo de estructuras comunes, en gran medida, a otros lenguajes tales como el Pascal.

C permite crear programas fáciles de modificar y de adaptar a nuevos ordenadores. Su «portabilidad» le hace el lenguaje idóneo para que programas en C escritos en un sistema puedan ejecutarse en otros con modificaciones mínimas.

Hoy día existen compiladores C para unos 40 sistemas, abarcando desde microprocesadores de 8 bits hasta el super-rápido

Cray1.

Otra característica a resaltar del C es su «potencia» y «flexibilidad». El sistema operativo UNIX está escrito, en su mayor parte, en lenguaje C. También existen cantidad de compiladores en el mercado desarrollados bajo C, entre los que podríamos mencionar Pascal, Lisp, Logo, Basic, etc.

Para los estudiosos de los diferentes lenguajes de programación diremos que C posee ciertas estructuras afines al Pascal, la sencillez del Basic y en ciertos aspectos llega a niveles tan bajos de la máquina que podemos decir que estamos trabajando con Ensamblador (¡casi nada!).

En resumen, las características principales a destacar del lenguaje C son :

— Producir programas reducidos (código compacto).
— Producir programas eficientes.
— Ser un lenguaje portátil, potente y flexible.
— Servir de base para el desarrollo de compiladores de otros lenguajes de programación.
— Aprovechar las ventajas de la programación estructurada.
— Ser de gran utilidad para estudiantes y profesionales.
— La instalación es económica.
— Herramienta óptima para el diseño.
— Utilización en áreas tan extensas como los paquetes software, la gestión, los minis y microordenadores y los juegos, entre otros.

LA ESCRITURA DE PROGRAMAS EN C Y SU COMPILACION

Los lenguajes de programación, en general, están basados en lenguajes «intérpretes» y lenguajes «compilados». Entre los primeros podemos citar el LOGO y el BASIC, perteneciendo al segundo grupo de los llamados compilados lenguajes, tales como el FORTRAN, PASCAL y C.

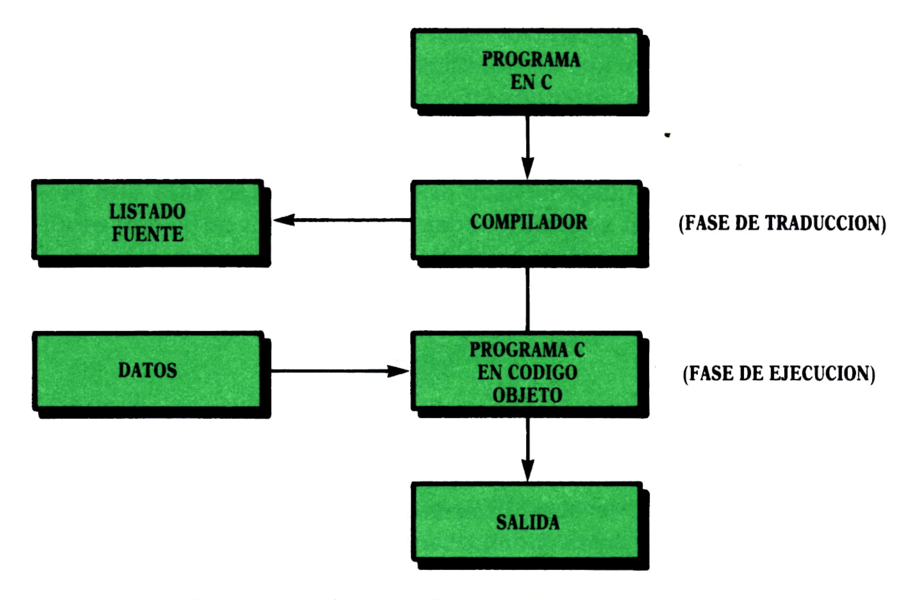


Fig. 3. Fases en la Compilación y Ejecución de un programa en C.

La primera fase para escribir un programa en C es disponer de un editor de propósito general, ya que el C no dispone de uno propio, para empezar a escribir su programa. Su ordenador dispondrá de un editor para realizar esta tarea. Si prefiere utilizar un editor distinto del que dispone su sistema, puede recurrir a procesadores de texto tan conocidos como el Wordstar utilizando la opción *N* (opción de «no documento»). Una vez escrito su programa, debe darle un nombre, recordando que debe terminar con *.c*, como, por ejemplo:

prog.c
hola.c

En un sistema UNIX el editor podría ser invocado tecleando **ed, ex, edit, emacs,** o **vi.** Los ordenadores personales disponen de editores tales como **edlin, ed, Wolkswriter,** etc.

Una vez escrito su programa, éste debe compilarse por medio del compilador de C, cuya misión es detectar si su programa posee algún error y, en caso de no existir ninguno, traducirlo al lenguaje que entiende la máquina, es decir, el código objeto. La traducción de su programa lo colocará en un nuevo fichero. Bastará invocar el nombre de este último fichero para que nuestro programa se ejecute.

Veamos un ejemplo:

P_t include <stdio.h>
main ()
printf («Este es mi primer programa en C»Ñn);

Este texto que hemos tecleado, gracias al editor, se guardaría en un fichero y se compilaría dando como resultado un fichero que al ser invocado ejecutaría nuestro programa.

En el sistema UNIX, el compilador de C se denomina **cc,** por lo que para compilar nuestro programa bastaría teclear

cc <nombre de mi programa>

Terminada la compilacion observaríamos que se ha creado un nuevo fichero llamado **a.out.** Tecleando dicho nombre nuestro programa se ejecutaría apareciendo en pantalla el siguiente texto:

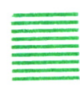
Este es mi primer programa en C

Para ordenadores personales existe una versión de compilador de C llamada Laticce C. Los pasos que daremos para ejecutar nuestro programa serán los siguientes:

mc1 <nombre de mi programa>
mc2 <nombre de mi programa>
link c <nombre de mi programa>

Este último paso originará un fichero llamado **nombre de mi programa.exe** y si a continuación tecleamos **nombre de mi programa.exe** conseguiremos ejecutar nuestro programa.

La estructura en general de un programa C sería:



ELEMENTOS DEL LENGUAJE C

El lenguaje C soporta los siguientes tipos de datos:

— Caracteres.
— Enteros.
— Números en coma flotante.
— Funciones.
— Arrays y punteros.
— Estructuras.
— Uniones.

Operadores, tales como:

— Aritméticos.
— Relacionales.
— Lógicos.
— Manejo de bits.
— Asignación.
— Condicional.
— Acceso a datos.

Estructuras de control, tales como:

— Sentencias.
— Bloques.
— Sentencia <<If-Then-Else>>.
— Sentencia <<switch>>.
— Saltos y etiquetas.

Tipos de constantes tales como:

— Enteros: dígitos numéricos.
— Números en coma flotante (caracteres y constantes alfanuméricas).

Funciones de entrada/salida, que forman parte de la librería estándar del C, y entre otras citaremos:

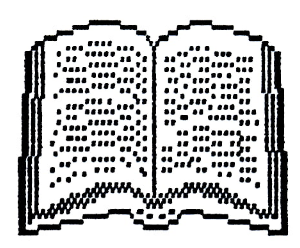


Fig. 4. C posee una librería estándar.

— getchar.
— printf.
— etc.

TIPOS DE DATOS, OPERADORES Y EXPRESIONES 2



N este capítulo estudiaremos la diferencia entre *variables* y *constantes,* cómo las variables deben *declararse* antes de utilizarse en un programa escrito en C, los diferentes tipos de datos reconocidos en C, los operadores y las expresiones como combinación de variables y constantes.

VARIABLES Y CONSTANTES

Tanto las variables como las constantes en un lenguaje de programación son datos que el programa utiliza para realizar una serie de tratamientos con ellos.

Ciertos datos se preseleccionan antes de la ejecución de un programa, manteniendo sus valores inalterables durante la misma, por lo que a dichos datos se les conoce como *constantes.*

Por el contrario, existen otros datos que pueden variar a lo largo de la ejecución del programa, llamándose a éstos variables. La diferencia entre datos variables y constantes es obvia. Mientras los primeros pueden cambiar su valor en el transcurso de la ejecución del programa, los segundos no sufren alteración.

Pongamos un ejemplo de todo esto. Supongamos una línea de programa C como la siguiente:

```
longitud = 2*3.1416*radio
```

En la expresión anterior tanto el dato 2 como el 3.1416 se consideran datos *constantes,* porque durante la ejecución del programa no cambiarán su valor. No sucede igual con *radio,* ya que si introducimos por el teclado de nuestro ordenador o a través de un bucle diferentes radios para calcular una serie de longitudes de circunferencias, el dato *radio* se modificará en el transcurso del programa, por lo que diremos que radio es una *variable.*

El nombre de una variable tiene ciertas restricciones. Veamos esto. Cualquier variable está formada por una serie de letras y dígitos en secuencia, pero debemos tener muy claro siempre que el primer carácter de comienzo de una variable debe ser una letra. Nombres de variables válidos pueden ser:

área
h54
juan_ana

Por el contrario, no podrán utilizarse para variables nombres tales como:

5tuyo
juan.ana

Queda claro en los ejemplos anteriores que el carácter de subrayado «_» puede utilizarse como una letra más y nos servirá como separador en nombres compuestos. El punto «.» no debe utilizarse para estos fines, ya que en C se utiliza para referenciar otros elementos del lenguaje que veremos posteriormente. Las letras mayúsculas y minúsculas son diferentes en el lenguaje de programación C. Se utilizan tradicionalmente las minúsculas para los nombres de variables y las mayúsculas para nombres de constantes.

Solamente los ocho primeros caracteres de un nombre de variable son significativos, aunque pueden utilizarse en número superior a ocho. Debemos aclarar que el sistema operativo puede limitar la longitud de símbolos, por lo que puede ocurrir que nombres de variables externas y nombres de función queden limitados a un máximo de siete caracteres. En cualquier caso, es conveniente siempre dar a la variable un nombre que tenga alguna relación directa con los objetivos de la variable dentro del programa.

Finalmente diremos que en C, como en todos los lenguajes de programación, existen *palabras reservadas* que no podrán utilizarse como nombres de variables. Tal es el caso de las siguientes palabras reservadas:

auto	break	case	char	continue
default	do	double	else	entry
enum	extern	float	for	goto
if	in	long	register	return
short	sizeof	static	struct	switch
typedef	union	unsigned	until	while

Como puede apreciarse, el número de palabras reservadas en C es inferior al de otros lenguajes de programación.



DATOS: SUS TIPOS

Como sabemos, cada vez adquiere más importancia el concepto de datos dentro de un programa. Uno de los clásicos errores en programación es asignar a una variable un tipo de dato cuando dicha variable pertenece a un tipo de datos diferente. Dentro del lenguaje de programación C los tipos de datos se clasifican en:

— *char*
— *int*
— *float*
— *double*

TIPOS DE DATOS	
CHAR	
INT	**SHORT**
	LONG
FLOAT	
DOUBLE	

Fig. 1. Tipos de datos.

El tipo *char* almacena en un byte (ocho bits) un carácter que corresponde al juego de caracteres empleado en su instalación (ASCII o EBCDIC).

El tipo *int* son enteros (sin parte decimal), que pueden tener signo o no. Una palabra de 16 bits puede almacenar un entero sin signo comprendido entre 0 y 65535, pero la misma palabra podría albergar un entero con signo entre −32768 y +32768. Para el tipo *int* existen una serie de calificadores tales como *short, long* y *unsigned.*

El calificativo *short* hace referencia a enteros de 16 bits, mientras que *long* se refiere a un doble entero. En cuanto al calificativo *unsigned* (sin signo), diremos que los números *unsigned* son siempre positivos. Puede utilizarse este tipo para estar seguros de que el valor de una variable nunca será negativo.

El tipo *float* representa un número en punto flotante y simple precisión. Este tipo de datos es utilizado en programas que poseen un gran número de cálculos matemáticos. Su equivalente en otros lenguajes de programación tales como Fortran o Pascal sería el tipo *real*. Por regla general, se utilizan 32 bits para almacenar un número en punto flotante. De estos 32 bits ocho son utilizados para representar el exponente y su signo, quedando 24 bits para expresar el valor de la parte no exponencial (mantisa). Este sistema permite una precisión de seis o siete cifras decimales y un rango de $\in$ (10^{-37} a 10^{+37}).

El tipo *double* (doble precisión) utiliza un número doble de bits, es decir, 64. Incorporando los 32 bits adicionales a la mantisa se consigue un mayor número de cifras significativas, alcanzándose una mayor precisión.



DECLARACION DE VARIABLES

Todas las variables en C, al igual que en Pascal, deben ser declaradas antes de su utilización. En todo programa C suministraremos una lista de variables que se utilizarán a lo largo del programa, indicando a qué tipo pertenecen.

Veamos un ejemplo de declaración de variables:

```
int a,b,c;
long int z;
char letra, silaba;
```

Como vemos, las comas se utilizan para separar los nombre de variables dentro de la declaración, empleándose el punto y la coma para finalizar la lista. Las variables de un mismo tipo pueden reunirse en una misma sentencia o utilizar varias. En el ejemplo anterior la declaración

```
int a,b,c;
```

podría haberse puesto de la siguiente forma:

```
int a;
int b;
int c;
```

En la declaración de variables podemos inicializar éstas a un determinado valor, como, por ejemplo:

```
int p = 75;
char palabra = 'p';
```

El ejemplo anterior definiría una variable entera con un valor inicial de 75 y una variable del tipo carácter con valor inicial *p*.



OPERADORES

<table>
<tr><th>OPERADORES</th></tr>
<tr><td>( ) [ ] - > •</td></tr>
<tr><td>! ~ ++ -- -(TIPO) * & SIZEOF</td></tr>
<tr><td>* / %</td></tr>
<tr><td>+ -</td></tr>
<tr><td><< >></td></tr>
<tr><td>< <= > >=</td></tr>
<tr><td>= = ! =</td></tr>
<tr><td>&</td></tr>
<tr><td>^</td></tr>
<tr><td>|</td></tr>
<tr><td>&&</td></tr>
<tr><td>| |</td></tr>
<tr><td>?:</td></tr>
<tr><td>= + = - = * = / = % =</td></tr>
<tr><td>,</td></tr>
</table>

Fig. 2.

En el lenguaje C existen los siguientes tipos de operadores:

— Aritméticos.
— Relacionales.
— Lógicos.
— Manipulación de bits.
— Asignación.
— Incremento y decremento.
— Condicionales.

Veamos cada uno de ellos.

— *Operadores aritméticos*

OPERADORES ARITMETICOS	
+	SUMA
-	DIFERENCIA
-	UNARIO
*	PRODUCTO
/	COCIENTE
%	RESTO
++	INCREMENTO
--	DECREMENTO

Fig. 3. Operadores aritméticos.

Los operadores aritméticos binarios son: + (suma), – (resta), * (multiplicación), / (división) y el operador módulo %.

Si realizamos una división entera de dos números, se truncará cualquier parte fraccionaria. Por ejemplo, la expresión

a % b

nos da el resto de dividir *a* entre *b* y será cero cuando *a* sea divisible entre *b*. El operador % no puede utilizarse con operandos del tipo *float* o *double*.

Todos estos operadores tienen un nivel de precedencia. La multiplicación y la división y el resto módulo (%) tienen mayor precedencia que la adición y la sustracción y, por tanto, se ejecutan antes. Si dos operadores tienen la misma precedencia se ejecutan según el orden en que aparecen en la sentencia de izquierda a derecha.

— *Operadores relacionales*

OPERADORES DE RELACION	
<	menor que
< =	menor o igual a
= =	igual a
> =	mayor o igual a
>	mayor que
! =	distinto de

Fig. 4. Operadores de relación.

Los operadores de relación en C son: > (mayor que), < (menor que), >= (mayor o igual) y <= (menor o igual).

Los operadores de igualdad son: == (igual) y != (distinto de).

En cuanto al orden de precedencia, o prioridad en la ejecución, todos los operadores relacionales tienen la misma precedencia, siguiéndoles en ella los operadores de igualdad.

— *Operadores lógicos*

OPERADORES LOGICOS	
&&	AND lógico
| |	OR lógico
!	NOT lógico

Fig. 5. Operadores lógicos.

Las conectivas lógicas son: && (and) y | | (or). Las expresiones donde aparecen ambas conectivas se evalúan de izquierda a derecha, interrumpiéndose la evaluación una vez conocido el resultado *true* o *falso.*

El operador && tiene un nivel de precedencia en ejecución superior al operador | |, y ambos tienen menor precedencia que los operadores relacionales.

— *Operadores lógicos para manejo de bits*

OPERADORES PARA EL TRATAMIENTO DE BITS	
&	AND
|	OR
^	OR EXCLUSIVO
<<	DESPLAZAMIENTO A LA IZQUIERDA
>>	DESPLAZAMIENTO A LA DERECHA

Fig. 6. Operadores para manipulación de bits.

Los operadores para la manipulación de bits no son aplicables a operandos del tipo *float* y *double.* Entre los operadores de manipulación de bits tenemos : & (and lógico), (or lógico), ˜(or exclusivo lógico), >> (desplazamiento a la derecha), << (desplazamiento a la izquierda) y ˜(complemento a uno o negacion de bits).

Los operadores lógicos de bits trabajan bit a bit, y pueden tratar cada bit independientemente.

Veamos algunos ejemplos de estos operadores:

(01110011) & (00100110) == (00100010)
(10011001) (10101010) == (10111011)
(10010011) ^(00111101) == (10101110)

— *Operadores de asignación*

OPERADORES DE ASIGNACION	
=	Asigna variable dcha. a variable izda.
+ =	Suma cantidad «d» a variable «i»
- =	Resta cantidad «d» de la variable «i»
* =	Multiplica la variable «i» por la variable «d»
/ =	Divide la variable «i» entre la variable «d»
% =	Resto de la división de «i» y «d»

Fig. 7. Operadores de asignación.

El operador de asignación más elemental es el =, que no debe confundirse con el operador == de comprobación de igualdad.

Expresiones como p = p + 1 pueden escribirse de la forma p += 1 utilizando el *operador de asignación* +=.

Lo anterior es válido a los siguientes operadores : +, -, *, /, %, <<, >>, &, ^, |.

En general, si *v1* es una cierta variable, *op* un operador y *e1* una expresión, entonces v1 op= e1 es equivalente a v1 = (v1) op (e1).

Veamos algunos ejemplos de todo esto:

z *= ++y es equivalente a
z = y + 1
z = z * y
s *= t + 1 es equivalente a s = s * (t+1)

Observemos en este último ejemplo la importancia del paréntesis, ya que es distinto s= s * t + 1 que la expresión de arriba.

— *Operadores de incremento y decremento*

Para incrementar y decrementar en una unidad a un operando el lenguaje de programación C dispone de dos operadores : ++ (le suma 1 a su operando) y -- (le resta 1 a su operando). Estos operadores pueden ir colocados antes de la variable o después de la variable. Dependiendo de la colocación, los efectos serán diferentes. Veamos algún ejemplo:

s = n++ incrementa n después de utilizar su valor. Si n es 7, asigna a s, el valor 7 y n se incrementa en 1 (n=8).

s = ++n incrementa primero n en una unidad y el resultado lo asigna a s. Si n es 7, asigna el valor 8 a s, quedando n con otro 8.
Algo similar produciría el operador --.

— *Operador condicional*

En C se puede abreviar la sentencia *if-else* tan conocida en otros lenguajes de programación, a través del operador condicional ?:. Pongamos un ejemplo:

x = (y < 0) ? -y : y;

El significado de la expresión anterior es la siguiente: si y es menor que cero, entonces x = - y; en caso contrario, x = y. Esto podría ponerse en forma *if-else*, así:

```
if ( y < 0 )
  x = - y;
else
  x = y;
```

La forma general de una expresión condicional es:

expresion1 ? expresión2 : expresión3.

Si expresión1 es cierta, la expresión condicional total toma el valor de la expresion2; si, por el contrario, expresion1 es falsa, toma el valor de la expresión3.
Finalmente, se estudiarán más adelante los «operadores relacionados con punteros» y los «operadores de estructuras y uniones».

OPERADORES RELACIONADOS CON PUNTEROS	
&	Operador dirección
*	Operador indirección

Fig. 8. Operadores relacionados con punteros.

OPERADORES DE ESTRUCTURAS Y UNIONES	
•	Pertenencia (especifica miembro)
->	Pertenencia indirecto

Fig. 9. Operadores de estructuras y uniones.

CONVERSIONES DE TIPO

Existen una serie de reglas para convertir operandos de diferentes tipos a un mismo tipo. Si en una expresión utilizamos variables de diferentes tipos, se llevará a cabo una reunificación de tipos según las siguientes reglas:

— Siempre será «ascendido» el tipo inferior al tipo superior antes de que se realice la operación.

— El rango de tipos de mayor a menor es:

double -> float -> long -> int -> short -> chart

Para los tipos *unsigned* poseen el mismo rango que el tipo al que están referidos.

Veamos algunos ejemplos:

```
int j;
char k;
j = k;
k = j;
```

En el ejemplo anterior el valor de K no se modifica.

Si y es del tipo *float* e i es *int, entonces:*

```
y = i;
i = y;
```

con estas dos asignaciones se produce una conversión; de *float* a *int* se provoca un truncamiento de la parte fraccionaria.

Veamos un pequeño programa donde aparecen estos conceptos:

```
main ()
{ ft = i = cr = 'X';
printf ("cr = %c, i = %d, ft = %2.2\n", cr, i, ft);
}
```

El resultado de las conversiones produce el siguiente resultado:

cr = X i = 88 ft = 88.00

A pesar de lo dicho en líneas precedentes, podemos forzar la conversión de un tipo dado mediante una construcción llamada *cast.* La forma general de dicho operador es:

(tipo) expresión

donde se sustituirá el nombre del tipo que queremos imponer en lugar de la palabra tipo.

Por ejemplo:

suma = 2.3 + 3.4 (suponemos suma de tipo int)

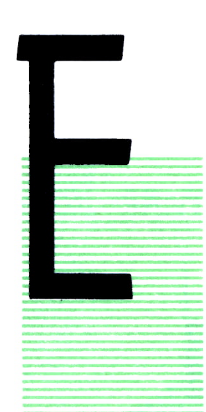
sumaría 2.3 con 3.4, dando 5.7 y truncando el resultado a 5 para convertirlo en tipo *int*.

En este otro caso:

suma = (int) 2.3 + (int) 3.4

2.3 y 3.4 se convertirían en 2 y 3 por los operadores (int), por lo que el valor asignado a *suma* sería igual a 5.

SENTENCIAS DE CONTROL 3



EN este capítulo estudiaremos los mecanismos que permiten que una sentencia o sentencias, escritas en C, se ejecuten una o varias veces, o se produzcan bifurcaciones en unas determinadas partes de un programa dependiendo de una condición.

En C debemos tener presente que, a diferencia del Pascal, el punto y coma utilizado al final de cada sentencia es un *terminador* de sentencia y no un *separador,* como ocurre con los lenguajes derivados del Algol.

El lector introducido en Pascal sabe que cualquier bloque de sentencias comienza con un *begin* y acaba con un *end.* Pues bien, en lenguaje C se utilizan las llaves } para agrupar un conjunto de sentencias o bloques.

Pongamos un ejemplo y su equivalente en Pascal:

Lenguaje C

```
while (j > 20)
{   z = j * j;
printf ("%d\n",z,j);}
```

Lenguaje Pascal

```
while j > 20 do
begin
z =: j * j;
writeln(z:5,j:5);
end
```



SENTENCIA IF-ELSE

En C, al igual que en Pascal, se utiliza la sentencia *if* para realizar bifurcaciones condicionales, es decir, tomaremos una decisión dependiendo de si una expresión es *cierta* o *falsa*.

La sintaxis empleada es:

```
if (expresión)
  sentencia;
```

La sentencia *if* nos permite ejecutar una sentencia, si se cumple una condición, como la contraria, a través de *else:*

```
if (expresión)
sentencia A;
else
sentencia B;
```

El *anidamiento* de varios *if* puede realizarse de la siguiente forma:

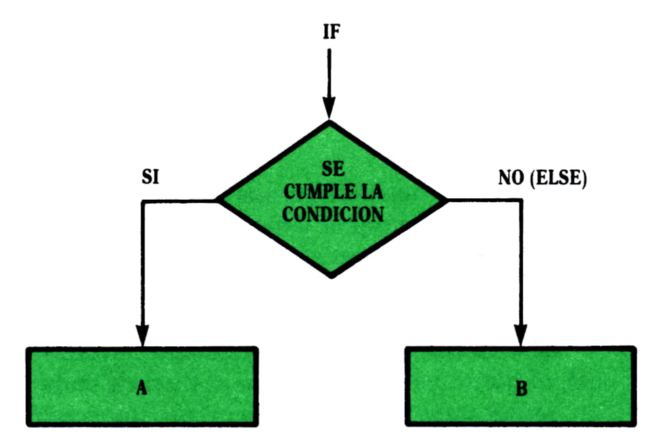


Fig. 1. Sentencia if-else.

```
if (a > 10)                              if (a > 10)
  if (b > c)                             if (b > c)
    z = b;                               z = b; }
  else                  else
    z = c;              z = c;
```

Ambas construcciones son lícitas, pero como vemos en el ejemplo de la parte derecha, se han utilizado las llaves para reforzar la asociación deseada. En el ejemplo de la izquierda se ha asociado la partícula *else* con el *if* más interno.



SENTENCIA SWITCH

La sentencia *switch* va acompañada de una expresión entre paréntesis. La sentencia evalúa la expresión y compara su valor con todos los casos *(case)*. Si alguno de los casos es idéntico al valor de la expresión que acompaña al *switch,* se ejecutará la sentencia etiquetada por *case.*

Veamos un ejemplo:

```
switch (v)
{case 'x':
printf ("esto es una x\n");
```

```
break;
case 'y':
printf ("esto es una y griega\n");
break;
default:
printf(«no es ni x ni y griega» \n»);
}
```

Como podemos apreciar en este ejemplo, la sentencia *break* se introduce para salir del *switch* y dirigirse a la sentencia que está inmediatamente después.

La línea con *default* se utiliza en el supuesto de que no se satisfaga ninguno de los *case* al compararse con la expresión que acompaña al *switch.*



SENTENCIA WHILE

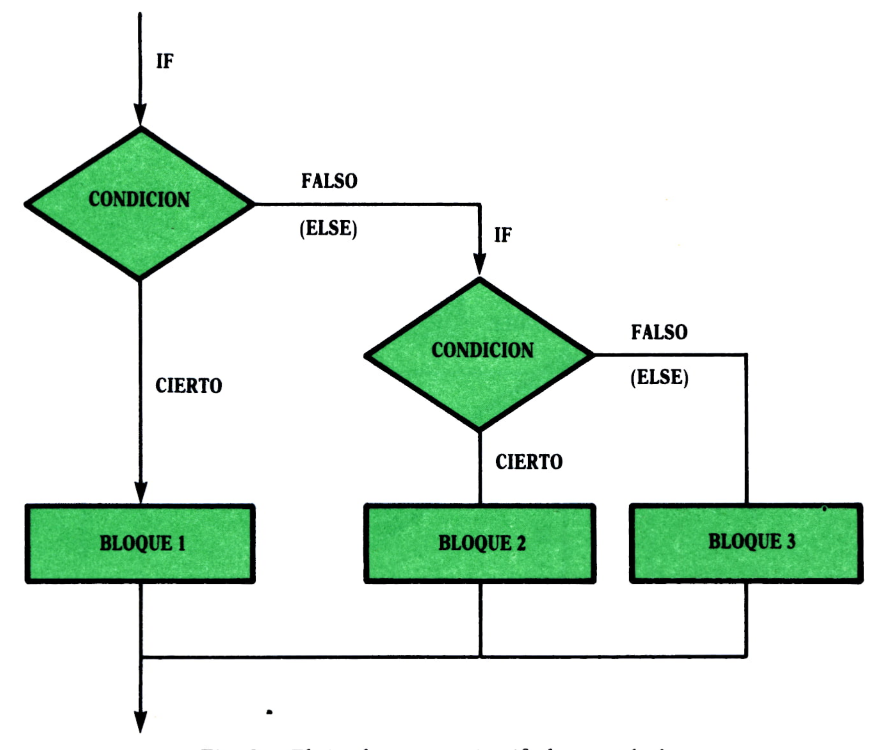


Fig. 2. Flujo de sentencias if-else *anidadas.*

Es una sentencia que se utiliza para la construcción de bucles. La repetición puede llevarse a cabo sobre una sentencia o bloque de senten-

cias. En el caso de utilizarse un grupo de sentencias dentro de una sentencia *while,* éstas deben ir encerradas entre *llaves.* La sintaxis de una sentencia *while* es:

```
while (expresión)
sentencia;
```

Se procede de la siguiente forma: se evalúa la expresión y si resulta ser cierta, se ejecuta la sentencia una y otra vez hasta que la expresión deje de ser cierta. Debemos tener la precaución de que la sentencia que estamos evaluando permita que en un momento dado la expresión se haga *falsa,* porque, de lo contrario, el bucle se repetiría indefinidamente.

Veamos un ejemplo de esta sentencia:

```
cuenta = 1;
while (cuenta < 3)
{printf("estamos contando\n");
cuenta = cuenta + 1;
}
```

Observe que si cuenta = 4, el bucle no se hubiese ejecutado por ser la condición falsa (cuatro no es menor que tres).



SENTENCIA FOR

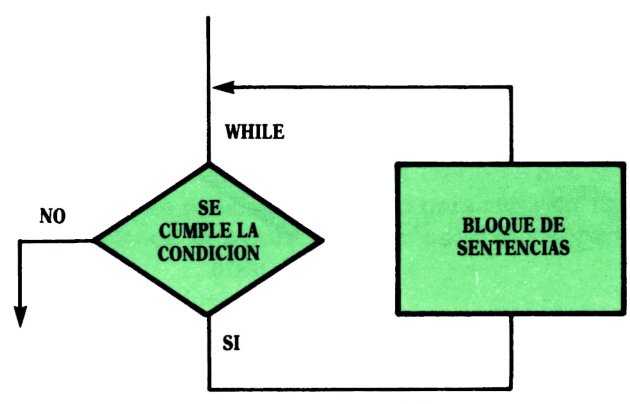


Fig. 3. Sistema while.

La sentencia *for* se utiliza para que una sentencia o bloque de sentencias se ejecute repetidamente en función de tres expresiones separadas por punto y coma.

La sintaxis general de un bucle *for* es:

```
for (expresión1; expresión2; expresión3)
sentencia;
```

La primera expresión es la de inicialización, la segunda de condición de prueba y la tercera se evalúa al final de cada bucle. Cuando la condición de prueba es falsa, damos por terminado el bucle. Veamos un ejemplo de la sentencia *for:*

```
main ()
{int numero;
for (numero = 1; numero <=3; numero++)
printf("%4d %4d\n", numero, numero*numero);
}
```

El pequeño programa anterior nos daría una tabla de todos los cuadrados de los números comprendidos entre 1 y 3 (inclusive). Como vemos, la sentencia se compone de un valor inicial (1), un valor final (3) y el incremento que sufre *número* en cada repetición del bucle.

Un hecho que debemos destacar con la sentencia *for* es la posibilidad de utilizar expresiones arbitrarias como componentes de la sentencia.



SENTENCIA DO-WHILE

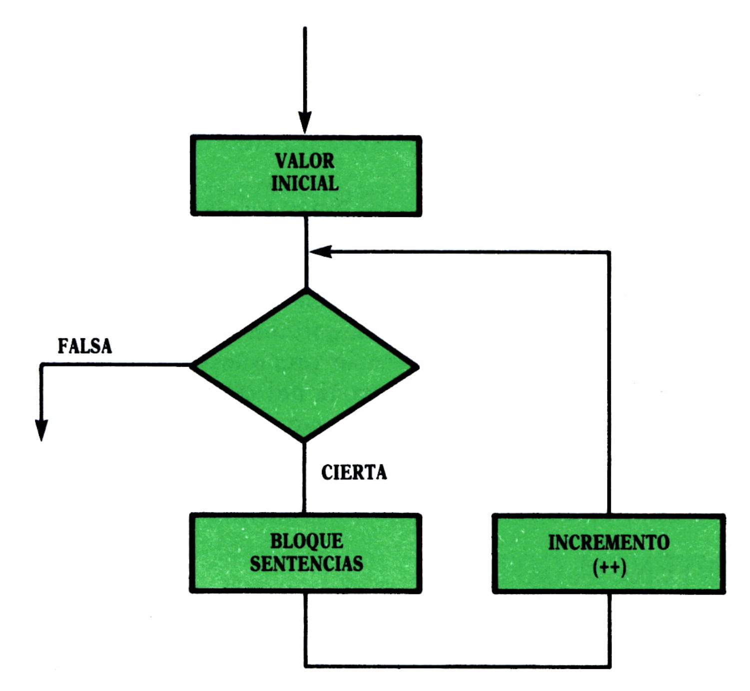


Fig. 4. Flujo de bucle for.

Tanto las sentencias *for* como *while* comprueban la condición de finalización del bucle al comienzo, como habrá podido comprobar el lector. Pero ¿qué ocurre si queremos ejecutar una sentencia por lo menos una vez antes de comprobar la condición? Para esto tenemos el tándem *do-while.* El bucle *do-while* comprueba la condición después de cada pasada a través de la sentencia o bloque de sentencias. Cuando la condición deja de cumplirse, el bucle finaliza.

La sintaxis utilizada es:

```
do
sentencia
while (expresión);
```

Veamos un ejemplo:

```
do
{printf("hola\n",a,c);
a = a + 1;
c = a;
}
while (a < 10);
```



SENTENCIAS BREAK Y CONTINUE

Esta sentencia nos permite, al igual que ocurría con la sentencia *switch,* la posibilidad de abandonar un bucle por un sitio distinto al de las comprobaciones de principio o final.

Esta sentencia fuerza la salida de un bucle *for, while o do,* de la misma forma que sucedía en *switch.*

La sentencia *continue* guarda relación con la sentencia *break,* pero su utilización es más restringida. Esta sentencia se utiliza en los tres tipos de bucles, excepto en *switch.* La sentencia *continue* hace que el resto de la iteración se ignore, obligando a comenzar una nueva iteración. La sentencia *continue* es de utilidad cuando la parte del ciclo que sigue es complicada y la posibilidad de introducir un nuevo anidamiento puede complicar en exceso el programa.



SENTENCIA GOTO

Los aficionados a la programación estructurada saben que esta sentencia es una sentencia *maldita* dentro de cualquier lenguaje que se precie de poder llamársele *estructurado.* Sin embargo, está presente en cualquier

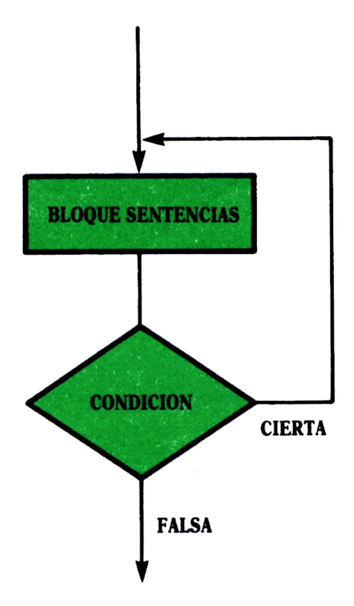


Fig. 5. Flujo de un bucle do while.

lenguaje de alto nivel, lo que no significa que tengamos necesariamente que utilizarla.

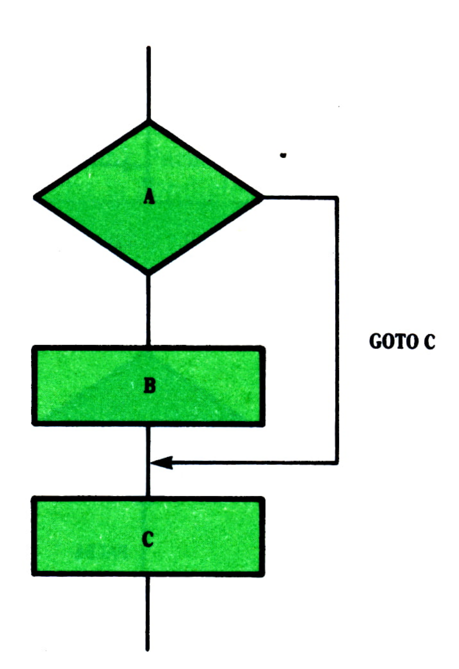
En lenguaje C, al igual que en Pascal, podemos prescindir de la sentencia *goto* casi por completo. La sintaxis de la sentencia es:

goto etiqueta

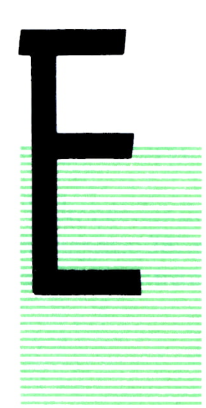
La etiqueta se rige por las mismas normas que las variables. El uso más normal de la sentencia *goto* consiste en salir de un proceso en una estructura que esté fuertemente anidada, ya que en este caso la utilización de la sentencia *break* sólo nos llevaría a abandonar la estrucutra más interna.

Veamos un ejemplo:

```
for (k = 0; k < S; k++)
for (m = 0 ; m < T; m++)
if (v[k][m] < 0)
goto a;
.................;
a : precio = 1.75 * unidad;
.................;
```


A
B
C
GOTO C

ESTUDIO DE FUNCIONES 4



ESTE capítulo lo dedicaremos a estudiar las *funciones* como filosofía de diseño del lenguaje C; aprenderemos cómo crear nuestras propias funciones y cómo se comunican.

La razón esencial para el empleo de las funciones es evitar bloques de sentencias que se repiten con cierta asiduidad dentro del cuerpo del programa. Todo programa escrito como un conjunto de funciones le da una característica de *modularidad* al programa, lo que redunda en una mejor comprensión y modificación de ciertas partes del programa.



COMO CREAR FUNCIONES

El C posee una serie de funciones definidas en el sistema tales como *printf(), scanf(), getchar(), putchar(), y strlen().* Nosotros podemos crear nuestras propias funciones, para lo cual estudiaremos el siguiente ejemplo: Supongamos que introducimos por el teclado de nuestro ordenador una serie de números de los cuales queremos obtener los números pares e imprimirlos. Para ello podíamos utilizar un programa como éste:

```
main ()
{int lista[30];
leernum (lista);
numpar(lista);
escribir(lista);
}
```

En el ejemplo anterior deberíamos escribir tres funciones: *leernum, numpar y escribir.* Como vemos, dentro del cuerpo principal del programa existen tres llamadas a tres funciones. Sólo necesitamos saber cómo lla-

marlas y a través de qué parámetros se establece la comunicación entre la función y el programa que la llama. Para empezar daremos la forma general que tiene una función:

```
tipo nombre (lista de argumentos)
declaraciones de argumentos;
{variables locales;
sentencia1;
sentencia2;
..........;
}
```

En el esquema anterior pueden suprimirse alguna de las partes o incluso todas, dando lugar a una función que no hiciese nada. Una función que no haría nada sería:

```
vacía ()
{ }
```

Veamos un pequeño programa realizado con funciones. Supongamos que deseamos imprimir una línea con 50 ceros (0). Veamos cómo podríamos hacer esto creando una función llamada *ceros:*

```
main ()
{  ceros();
}
/* funcion ceros */
 #include <stdio.h>
 #define MARGEN 30
ceros ()
{ int contador;
for  (contador = 1; contador <= MARGEN; contador++)
      putchar ('0');
      putchar('\n');}
}
```

Ejecutado el programa, la respuesta sería :

000000000000000000000000000000

Estudiemos cómo se ha desarrollado el programa. En primer lugar, se ha *llamado* a la función *ceros* en el programa *main* tan sólo escribiendo su nombre. Esta llamada provoca el salto a la función *ceros* y ejecuta las instrucciones que contiene dicha función, para, una vez terminadas todas ellas, volver a la siguiente línea de programa del *programa de llamada.*

Si examina con atención el programa habrá observado que la función *ceros* sigue la estructura que definimos en líneas anteriores para una función en general, es decir, la función *ceros* comienza con una cabecera, en este caso instrucciones de preprocesador #include, #define) y nombre de

la función, y un cuerpo de función que contiene una sentencia de declaración (int contador) y el resto de sentencias que componen la función.

Como habrá observado, el nombre de una función nunca va seguido por punto y coma. Deberá tener en cuenta este hecho para saber que está trabajando con una función y no con una sentencia.

La función *ceros,* en nuestro ejemplo, no necesitó ningún *argumento de entrada,* por lo que no existió comunicación entre el programa de llamada y la función. Estudiaremos el caso de funciones con argumentos en el próximo apartado.



FUNCIONES CON ARGUMENTOS

Recordando de nuevo la estructura general dada en el apartado anterior, para definir una función iremos recorriendo cada uno de los aspectos de que está compuesta:

```
tipo nombre (lista de argumentos)
declaraciones de argumentos
{variables locales;
sentencia1;
sentencia2;
..........;
}
```

Tipo nombre se refiere al tipo de dato que devuelve la función. Por defecto, se supone que es de tipo *int.*

Lista de argumentos es una relación de nombres separados por comas. Puede no existir ningún argumento (caso de la función *ceros),* pero sigue haciéndose necesario el empleo de ().

En las declaraciones de argumentos declararemos las variables utilizadas como argumentos de la función. Por defecto, se suponen que son del tipo *int,* caso de no declararse ninguna.

Variables locales son aquellas que sólo tienen validez dentro del propio cuerpo de la función, es decir, utilizadas en el programa principal o en otras funciones harían referencia a variables diferentes.

Para devolver un valor al programa de llamada desde una función utilizaremos la palabra clave *return.* Con esta palabra se da por terminada la ejecución de la función, devolviendo control a la siguiente línea de la función de llamada.

Veamos un ejemplo de todo lo explicado hasta aquí. Supongamos que queremos hallar el área de un rectángulo. Para ello deberemos conocer sus dos lados x(base) e y(altura).

Trataremos de hacer un programa en C que a través de una función área, calcule el área de un rectángulo de base = 4 y altura = 2.

```
main()
{rect int;
rect = area (4,2);
printf("rect= d\n",rect);
/* función área*/
area(b,h)
{int producto;
producto = b*h;
return (producto);
}
```

Expliquemos un poco todo el proceso seguido. Al llamar a la función con área(4,2) se crean dos nuevas variables con los nombres b y h. Los números cuatro y dos, utilizados en la llamada a la función *área,* son parámetros que se asignan a los argumentos *b* y *h* de la función. A continuación se ejecutan las sentencias del cuerpo de la función, finalizando con la sentencia *return* (producto), que da como resultado que el valor de la *función area(4,2)* sea el valor de la variable *producto* (en nuestro caso 4*2=8). El programa sigue ejecutándose en la siguiente línea a la llamada a la función, es decir, se imprime el resultado del área calculada.

VARIABLES EXTERNAS Y ALCANCE DE UNA VARIABLE

En el lenguaje de programación C, a diferencia del Pascal, no podemos definir funciones dentro de otras. Las *variables externas* están definidas fuera de la función y pueden ser compartidas por otras, es decir, son variables *globales* y como tales son análogas al COMMON de FORTRAN, por citar un ejemplo.

Si la variable externa se ha definido, cualquier función podrá acceder a ella con tan sólo referenciarla, evitando en algunos casos grandes listas de argumentos.

Si la variable externa se declara dentro de la función que la utiliza, se deberá declarar con la palabra clave *extern.*

Veamos un ejemplo.

```
int ejemplo;
main()
{extern int ejemplo;
.................;
.................;
}
```

```
hola ()
{extern int ejemplo;
.................;
}
```

En el ejemplo anterior hemos creado una variable externa *ejemplo*, conocida tanto por main() como por la función *hola*.

El alcance o validez de una variable externa es su permanencia, es decir, permanecen en el ordenador durante la ejecución del programa, y al no ser de ninguna función en particular, no quedan eliminadas al finalizar alguna de ellas.



RECURSIVIDAD

En C, así como en PASCAL, las funciones pueden utilizarse de modo recursivo. El proceso por el cual una función puede llamarse a sí misma se conoce con el nombre de *recursividad*.

No confundamos que una función pueda llamarse a sí misma con los bucles *while* o *do while*. Cuando una función se llama a sí misma, cada invocación originará la creación de una copia de todas las variables, independiente de la anterior, por lo que la utilización de la memoria será superior y el tiempo de proceso por parte del ordenador será más lento, pero esta propiedad de las funciones tiene como ventaja la claridad en la escritura y su sencillez.

Veamos un ejemplo de una función recursiva:

```
main ()
printf("Hola, estoy repitiéndome continuamente\n");
main ();
}
```

Como vemos en el ejemplo anterior main () se llama a sí misma.

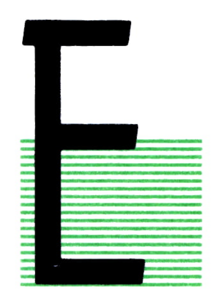
COMPILACION DE PROGRAMAS CON VARIAS FUNCIONES

La compilación de varias funciones que están en el mismo fichero se lleva a cabo de la misma forma que la compilación de una sola función.

Dependiendo del sistema, la compilación se realizará de forma diferente; por ejemplo, en el sistema operativo UNIX el comando *cc prog1.c prog2.c* (suponiendo *prog1.c* y *prog2.c* ficheros que continen funciones) producirá un ejecutable llamado *a.out*. En otros sistemas tales como *Lattice C*, compilando por separado *prog1.c* y *prog2.c* se producirán dos ficheros objetos *prog1.obj* y *prog2.obj*. Estos módulos serán combinados, a través del linker, con los módulos objeto estándar de *c.obj:*

```
link c prog1 prog2
```

EL PREPROCESADOR C Y LOS MODOS DE ALMACENAMIENTO 5



EN este capítulo describiremos brevemente las funciones básicas cubiertas por el preprocesador del C, una valiosa herramienta que ayuda enormemente a los programadores gestionando ficheros y descargando a los usuarios de pesadas tareas.



SENTENCIAS DEL PREPROCESADOR

El preprocesador de C permite preprocesar macros e incluir el contenido de otros archivos durante la compilación. Una de las sentencias más comunes del preprocesador para la inclusión de un fichero de texto dentro de un programa es # *include*. Este comando, como el resto de comandos del preprocesador, debe comenzar con el símbolo #.

El formato de este comando es:

—# include «nombre del fichero»: el preprocesador realizará una búsqueda del fichero especificado por «nombre del fichero», incluyéndolo dentro de un programa. Dependiendo del sistema en el que estemos trabajando, la utilización de <> o comillas indicarán al preprocesador que realice la búsqueda en un directorio estándar o en su directorio, es decir, *#include <nombre del fichero>* obligará al preprocesador a iniciar la búsqueda del fichero en un directorio estándar.

La sustitución de macros (llamamos *macros a* la palabra que deseamos definir) se realiza a través del comando *#define,* seguida de la macro que queremos definir y de un texto que reemplazará a la macro cada vez que ésta aparezca en el programa. El formato del comando es:

—# *define macro texto:* esta definición lleva a cabo la sustitución de macro por texto cada vez que encontremos macro a lo largo del programa. La macro es un nombre que sigue las mismas reglas que las variables en C.

Veamos algunos ejemplos de estos dos comandos.

```
# define VOCALES «aeiou»
# define NUMERO 50
# define ANIMAL «leon»
# define SUMA NUMERO+NUMERO
main
{int y = NUMERO;
printf(y);
printf (ANIMAL);
y = SUMA;
printf(y);
printf (VOCALES);
}
```

La ejecución del programa daría como resultado:

```
50
leon
100
aeiou
```

Podemos ver con este sencillo ejemplo cuál es el efecto del comando *P,define* y cómo se produce la sustitución de la macro por el texto.

En realidad, existen pocas restricciones gramaticales sobre lo que puede definirse, por lo que los partidarios de Pascal podrían escribir algo parecido a esto:

```
# define then
# define begin{
# define end ;}
y a continuación
if (j < 10)
then
    begin
    p = 10;
    q = 15;
    end
```

Al trabajar con macros debemos tener la precaución de que la macro aparezca en la sentencia *printf* sin comillas, en el caso de que queramos imprimir los caracteres que representan. En el primer ejemplo definimos *#define ANIMAL «leon»* y dentro del programa existe una sentencia *printf (ANIMAL)*. Observe que ANIMAL va dentro de la sentencia sin comillas, ya que la utilización de éstas llevaría consigo que se hubiese imprimido *ANIMAL* y no *la palabra león* como pretendíamos.

Otro estupendo recurso al alcance del programador es utilizar *argu-*

mentos con la sentencia *#define*. La idea de argumentos ya se estudió cuando vimos las funciones, pero ahora vamos a utilizarlos con el comando *#define*.

Veamos un ejemplo:

```
# define SUMA(x) x+x
# define TEXTO(x) printf(«x es %d.ñn»,x)
main()
{int x = 5
int y;
y = SUMA(x);
TEXTO(y);
}
```

El programa ejecutado daría como resultado:

y es 10.

Una advertencia para no cometer errores cuando se están usando argumentos con macros. Debemos tener mucho cuidado con los paréntesis para estar seguros que se conservan los órdenes de evaluación. Basta poner un ejemplo para comprender lo dicho:

```
# define potencia(x) x * x
```

Cuando la macro sea llamada en un programa como potencia (y + 2) se evaluará como z = y + 2* y + 2, cuyo resultado es z = 3* y + 2 es completamente distinto al pretendido, que sería z = (y + 2) * (y + 2).



COMANDOS PARA PROGRAMAS DE GRAN TAMAÑO Y OTROS TIPOS

Otros comandos, menos utilizados que #define y #include, para compilación condicional y creación de nuevos nombres de tipos de variables, son:

#undef, #if, #ifdef, #ifndef, #else, #endif

Veamos como actúan cada uno de ellos:

— El comando *#undef* seguido de una macro borra la última definición de la macro. Veamos un ejemplo:

```
# define TOTAL 10
# define SUMA 20
# undef TOTAL
```

La última línea nos dice que la macro TOTAL queda sin definir.

— *# if constante - expresiòn* comprueba si la constante - expresión evaluada es distinta de cero.

— *# ifdef identificador* comprueba si el identificador ha sido definido de forma correcta por el preprocesador, en cuyo caso se ejecutarán el resto de comandos hasta que aparezca el primer *# ese o # endif*. Veamos un ejemplo:

```
# ifdef CAMELLO
# include «animal.h»
# else
# include «hombre.h»
# endif
```

En este ejemplo el comando *# ifdef* no indica que si CAMELLO ha sido definido por el preprocesador entonces se ejecutarán todos los comandos hasta encontrar el primer *# else*, si CAMELLO no ha sido definido se ejecutarán los que se encuentren desde # else hasta *# endif*.

— *# ifndef identificador* comprueba si el identificador no está definido.

Las construcciones anteriores pueden anidarse obteniendo estructuras más complejas.

El C permite crear nuevos nombres de tipos de variables llamados *tipos definibles*. Es el programador el que creará el tipo con un nombre definido por él. Para la creación de estos tipos el programador utilizará la palabra clave *typedef*.

Veamos un ejemplo:

```
typedef int ANCHO;
```

La línea anterior hace que ANCHO sea un nuevo nombre de tipo *int*.

Con respecto a *typedef* debemos aclarar que esta palabra no está creando tipos nuevos, sólo está cambiando de nombre a los tipos de datos.



TIPOS DE ALMACENAMIENTO

En el lenguaje de programación C cada variable tiene su tipo, como hemos visto en capítulos anteriores. Cada variable en C tiene su modo de almacenamiento y dependiendo de éste se controlarán las funciones que tienen acceso a esas variables y el tiempo que va a residir en memoria la variable.

Existen cuatro modos de almacenamiento en lenguaje C. Estos son: *auto (automático), static (estático), extern (externo) y register (registro).*

Veamos cada uno de ellos:

— *Almacenamiento auto (automático)*

Las variables automáticas están limitadas al ámbito local de la función, es decir, sólo ésta la reconoce porque han sido declaradas dentro de la función.

Cuando una función es invocada la variable dinámica toma conciencia de su existencia, pero una vez que ha terminado el «trabajo» de la función, la variable desaparece, por lo que su valor no se mantiene entre dos invocaciones sucesivas de la función. En resumen, el alcance de una variable automática queda limitado al bloque en el cual está declarada la variable.

TIPOS DE ALMACENAMIENTO		
AUTOMATICO	DURACION TEMPORAL	ALCANCE LOCAL
ESTATICO	DURACION PERMANENTE	ALCANCE LOCAL
EXTERNO	DURACION PERMANENTE	ALCANCE GLOBAL
REGISTRO	DURACION TEMPORAL	ALCANCE LOCAL

Fig. 1. Tipos de almacenamiento.

Ejemplo:

```
 main ()
{auto int minuto;
 ..................}
```

— *Almacenamiento static (estático)*

En este modo de almacenamiento las variables así definidas permanecen en memoria. Cada llamada a la función utiliza la misma posición de memoria, y a pesar de tener el mismo alcance que las variables automáticas, las estáticas no desaparecen cuando la función termina, sino que conservan su contenido entre dos invocaciones sucesivas de la función.

Ejemplo:

```
  main ()
{..........
 ..........
 }
 prueba ()
 {static int hora = 12
 ..........}
```

— *Almacenamiento externo*

Como su propio nombre indica, toda variable definida fuera de la función se conoce con el nombre de *externa*. Si la variable se declarase dentro de una función necesitaría la palabra *extern*. La utilización de la palabra *extern* nos permitirá que otras funciones puedan acceder a dicha variable. En el capítulo IV ya hicimos mención a este tipo de variables.

— *Almacenamiento tipo register (registro)*

Las variables registro son análogas a las de tipo automático, con la diferencia de que las primeras quedan almacenadas, si existe la posibilidad, en los registros internos del ordenador. Como podemos comprender, el acceso a dichos registros será mucho más rápido que en el caso de que la variable estuviese almacenada en memoria. Como no siempre es posible almacenar la variable en un registro del ordenador, en el caso de darse esta circunstancia la variable quedaría almacenada como variable automática.

Ejemplo:

```
main ()
{register int casa;
.................
}
```



INICIALIZACION

Si no especificamos explícitamente la inicialización, las variables externas y estáticas tendrán valor cero, pero no así las variables automáticas y registro, cuyo valor será indefinido, es decir, pueden contener cualquier cosa.

Para inicializar una varible basta con seguir el siguiente formato:

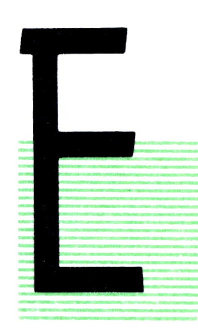
nombre de la variable = expresión constante

Por ejemplo:

```
int z = 100;
long minutos = 60 * 60;
```

ARRAYS Y PUNTEROS 6

¿QUE ES UN ARRAY?



L array es una estructura de datos formado por elementos del mismo tipo y utilizados para almacenar una serie de datos que nuestro programa necesita. Por ejemplo, podremos guardar las vocales en un array compuesto por cinco elementos.

Veamos algunas formas de declarar arrays.

```
int b[20];
static int vector[50];
exter int meses[12];
```

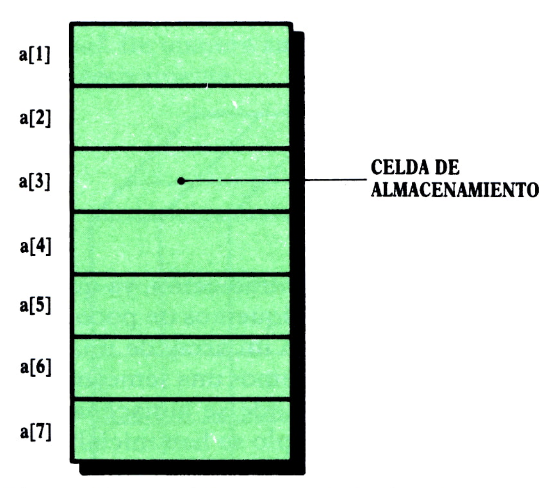


Fig. 1. Array unidimensional de siete elementos.

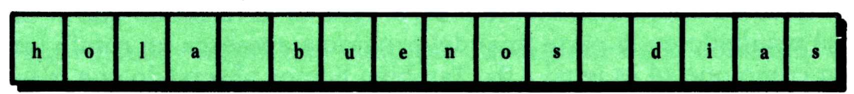


Fig. 2. Array de caracteres.

Como vemos, en las líneas anteriores se han definido tres arrays. El primero es un array automático, el segundo un array estático y el tercero un array externo. Las diferencias entre ellos podemos resumirlas diciendo que los arrays del tipo *extern* y *static* se inicializan a cero si no se les asigna ningún valor, mientras que los del tipo *automatic* y *register* si no son inicializados por nosotros pueden contener cualquier «cosa», dependiendo de la zona de memoria donde se les haya reservado el «hueco». En unas ocasiones será bueno tener inicializados los arrays y en otras será más práctico partir de un array con los elementos inicializados a cero, dependerá del programa que queramos construir.

Podemos comprobar si el número de elementos del array es inferior al tamaño del array que definamos, y además el array se declara como *extern*, entonces aquellos elementos que faltan serán inicializados a cero.

Veamos esto más claramente con un ejemplo:

```
int igual [4] = 1,2,3
main ()
{int indice;
extern int igual
for (indice = 0; indice < 4; indice ++)
printf ("El %d es igual al %d.\n", indice + 1, igual [indice]);
}
```

La ejecución del programa daría:

El 1 es igual al 1.
El 2 es igual al 2.
El 3 es igual al 3.
El 4 es igual al 0.

Bien, aquí parece existir una contradicción, ya que todos sabemos que el 4 nunca puede ser igual al 0. Expliquemos un poco el desarrollo del programa. Como vemos, se ha definido un array de una dimensión, llamado *igual,* con cuatro elementos y si hacemos una semejanza diríamos que cada elemento es como una «casilla» donde se almacenarán los datos en elementos. Pues bien, en nuestro ejemplo se han inicializado tan sólo las tres primeras «casillas» del array con los valores 1, 2 y 3, pero la «casilla» cuatro no ha sido inicializada, por lo que al ser el array de tipo *extern* esta «casilla» quedará automáticamente a cero. Esta es la razón de que al imprimir el resultado en la parte final del programa aparezca un rótulo tan absurdo como que «4 es igual a 0». También hemos observado que la inicialización de un array se debe hacer entre }.



ARRAYS MULTIDIMENSIONALES

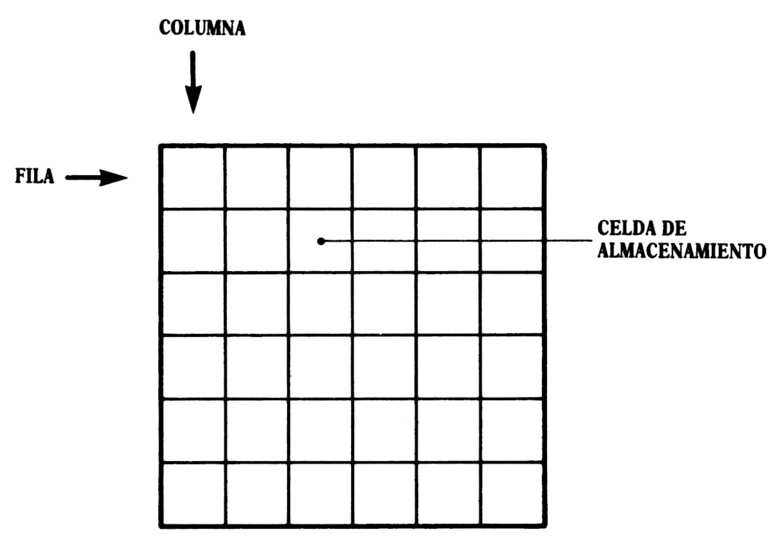


Fig. 3. Array bidimensional.

Los arrays vistos anteriormente se denominan *arrays unidimensionales* o *vectores*. Un array cuyos elementos sean arrays unidimensionales se llama *array bidimensional.* De manera análoga podríamos definir un array tridimensional, y así sucesivamente.

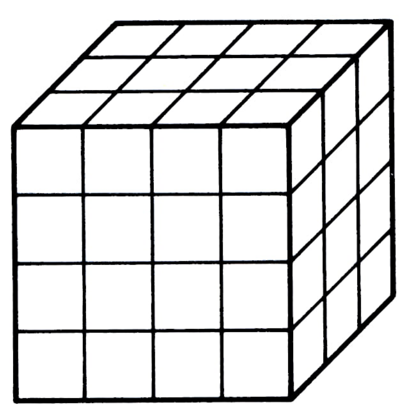


Fig. 4. Array tridimensional.

Para entender mejor este concepto supongamos que definimos el siguiente array:

```
double b[3][3];
```

Este array queda definido como una matriz de 3×3 elementos en coma flotante y doble precisión. La declaración en Pascal, para los amantes del Pascal, sería:

```
var
b : array [1..3,1..3] of real;
```

Veamos un ejemplo más complejo:

```
main()
{int v[4][4];
int i,j;
for (i = 0; i < 5; i ++)
     for (j = 0; j < 5; j ++)
     v[i][j] = 0;}
```

El programa inicializa un array bidimensional *v¡4ñ¡4ñ*, de 4×4 elementos a cero. Podemos observar el anidamiento de sentencias *for* para conseguir recorrer para una fila dada (i) las cinco columnas (j).



UNTEROS

Si en la literatura especializada usted encuentra en alguna ocasión la palabra «pointers», tradúzcala por *puntero,* pues ese es su significado; pero dejando aparte la lingüística, diremos que un *puntero* es un tipo de variable cuyo contenido es una dirección de memoria, donde hay un dato que nos puede ser de interés en un momento dado.

El operador que nos proporciona un puntero a un variable es el *operador &*. Por ejemplo, si «y» es una variable, un puntero a dicha variable es &y.

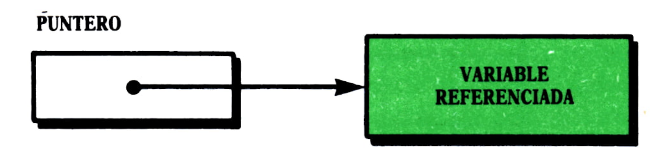


Fig. 5. Puntero a la variable referenciada.

La teoría de *punteros* puede parecer complicada y confusa para los principiantes en este tipo de variables (algo parecido ocurre cuando utilizamos muy a menudo la «proscrita» sentencia «goto»), pero siendo ordenados en nuestros planteamientos se pueden emplear para conseguir una mayor claridad y simplicidad de nuestros programas.

En C también existen *variables puntero,* es decir, no sólo se asigna a

una variable *int* un entero o a una variable *char* un carácter, también a una variable *puntero* se le podrá asignar como valor una dirección.

Por ejemplo:

```
punt = & vect;
```

asignará la dirección de *vect* a *punt,* o lo que es igual, se dice que *punt* está apuntando a *vect.*

El *operador indirección* * accede a la variable apuntada por un determinado puntero. Por ejemplo:

```
int a,b;
a = 2;
b = * (&a) + 1;
es similar a :
int a,b;
a = 2;
b = a + 2;
```

Veamos cómo debemos declarar en un programa los punteros:

```
int *pa;
float *pf;
```

En las declaraciones anteriores, la primera nos dice que «pa» es un puntero y que «*pa» es de tipo entero, es decir, el valor (*pa) apuntado por «pa» es de tipo entero. Algo análogo sucede con *float *pf,* excepto que es un puntero a una variable *float.*

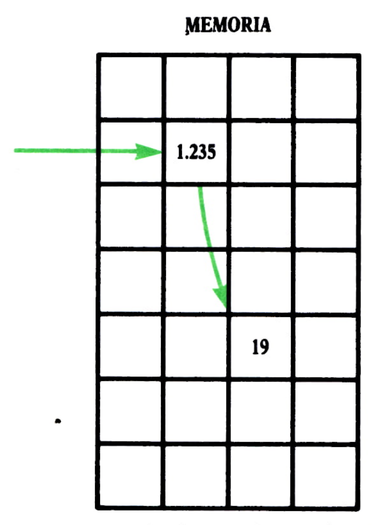


Fig. 6. Puntero que contiene la dirección (1235) de la variable referenciada.



PUNTEROS Y FUNCIONES

En el capítulo dedicado a funciones se estudió cómo se pasan argumentos a éstas. Las variables que le han sido pasadas a la función no pueden ser modificadas por la propia función. Esto no ocurre así en Fortran o Pascal. Veamos un ejemplo. Consideremos la función de «intercambio» que altera (intercambia) los valores de dos variables:

```
main ()
{int u,v;
u = 1;
v = 2;
printf("u = d% v = d%\n",u,v);
intercambia (&u,&v);
printf("u = d% v = d%\n",u,v);}
intercambia (px,py)
int *px, *py;
{area = *px;
*px = *py;
*py = area;
}
```

Este programa hace lo siguiente: En principio, los valores de *u* y *v* son uno y dos, respectivamente. Al final del programa los valores de u y v se han intercambiado, es decir, *u = 2* y *v = 1*. La función de llamada *intercambia(&u,&v)* envía direcciones, por lo que los argumentos *«px»* y *«py»* tendrán valores de direcciones y como tales tendrán que declararse como punteros a enteros. Como **px* nos da el valor de «u» y **py* nos da el valor de «v», la asignación **px = *py* es lo mismo que *u = v*, los valores se intercambian a través de las últimas líneas de la función «intercambia».

En Pascal se pasan parámetros por «valor» y parámetros por «variable». Estos últimos se devuelven modificados al procedimiento de llamada. La utilización de punteros para comunicación entre funciones permite alterar las variables de forma parecida a como se modifican los parámetros por «variable» del Pascal.



PUNTEROS Y ARRAYS

En C existe una estrecha relación entre punteros y arrays lo suficientemente fuerte como para que se los trate simultáneamente. Cualquier tratamiento que podamos hacer con arrays es susceptible de poderse hacer con punteros, pero consiguiendo una mayor rapidez con estos últimos, aunque su comprensión será más difícil.

Supongamos un array llamado «tabla»:

tabla = = &tabla[0]

La igualdad anterior se cumple porque tanto «tabla» como &tabla[0] representan la dirección de memoria del primer elemento. Ambos términos no podrán cambiar su valor, pero pueden ser asignados a una variable «puntero» y podemos modificar su valor. Veamos otro ejemplo. Supongamos la siguiente declaración:

int b[20]

con lo que estamos definiendo un array de tamaño 20, es decir, un bloque con 20 «casillas» tales como b[0], b[1], b[2], ... b[20]. Si utilizamos la notación b[i] para indicar el elemento i-ésimo del array y además «pa» es un puntero a un entero declarado como

int *pa

la asignación

pa = &b[0]

hace que «pa» apunte al elemento cero del array «b»; es decir, «pa» contiene la dirección de b[0]. Entonces:

y = *pa

copiará el contenido de b[0] en «y».

Si «pa» apunta a un elemento determinado de un array «b», entonces por definición «pa + 1» apunta al siguiente elemento, y en general «pa – i» apunta a «i» elementos anteriores a «pa» y «pa + i» apunta «i» elementos después.



ARRAYS Y FUNCIONES

Los arrays, al igual que sucede en Pascal, pueden aparecer como argumentos de funciones. Si intentamos pasar el nombre de un array como argumento de una función lo que realmente estamos pasando es un puntero, y esta circunstancia es aprovechada por la función para realizar los cambios necesarios en el array.

Por ejemplo:

```
cadena(a)
char a[];
{.........
.........
}
```

es lo mismo que escribir

```
cadena (a)
char *a;
{.........
.........
}
```

Veamos este otro ejemplo. Supongamos que deseamos pasar el array «máscara» como un argumento a la función «disfraz»:

```
disfraz(máscara)
char máscara[a][b];
{.........
.........
}
```

o también:

```
disfraz(máscara)
char máscara [][b];
{.........
.........
}
```

o por qué no ésta:

```
disfraz(máscara)
char (*máscara)[b];
{.........
.........
}
```

y en esta última vemos que «máscara» es un puntero a un array.

Se nos presentan dos alternativas para realizar una tarea, es decir, podemos utilizar arrays o punteros. Tan sólo diremos que los punteros presentan una mayor potencia, pero para los programadores menos experimentados es aconsejable el uso de los arrays por su mayor sencillez.



OPERACIONES BASICAS CON PUNTEROS

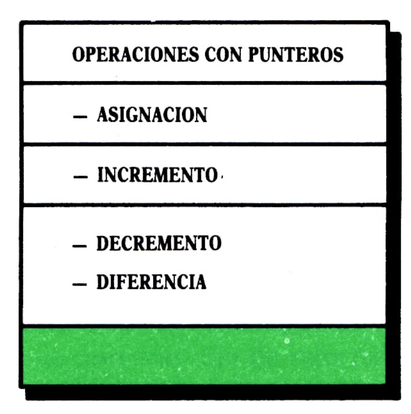


Fig. 7. Operaciones con punteros.

Con punteros podemos realizar cinco operaciones:

— *Asignación*

Siempre podemos asignar una dirección a un puntero.

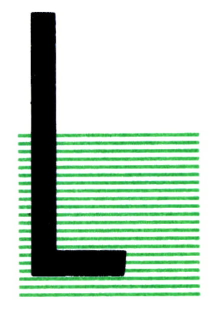
— *Incremento*

Un puntero puede ser incrementado utilizando el operador de incremento. De forma similar podemos decrementarlo.

— *Diferencia de punteros*

Es posible hallar la diferencia de dos punteros que apuntan al mismo array para averiguar el número de elementos que existen entre ellos.

ESTRUCTURAS EN C 7



LAS estructuras de datos las definiremos, en principio, como un conjunto fijo de información relativo a un solo objeto, refiriéndonos en unas ocasiones a la información y en otras a una parte de esa información.



DEFINICION DE UNA ESTRUCTURA

Una «estructura» es un formato de datos, flexible y capaz de representar una gran cantidad de datos diferentes. Los amantes del Pascal saben que una estructura de datos llamada «registro» está compuesta por un número fijo de componentes, llamados «campos», donde cada campo viene definido por su tipo e identificador. Pues bien, en C una «estructura de datos» está representada por un conjunto de variables, del mismo tipo o diferentes. Las componentes de una estructura se denominan en C *miembros*. De las líneas anteriores se desprende la similitud de concepto que existe entre el «registro» de Pascal y la «estructura» de C.

Una «estructura» puede declararse con la palabra clave «struct», seguida del «nombre de la estructura» y un conjunto de miembros y declaraciones encerrados entre llaves.

Veamos un ejemplo:

```
struct editor
{char nombre[15];
char direcc[40];
long telefono;
}
```

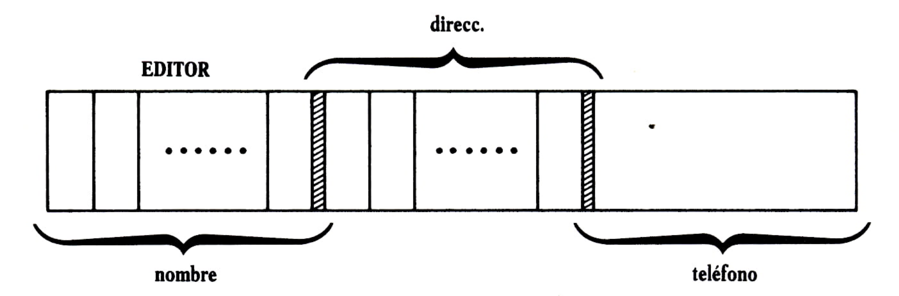


Fig. 1. Representación de una estructura.

En el ejemplo anterior «editor» es el nombre de la estructura y «nombre» «direcc» y «teléfono» son miembros de distinto tipo.

Podemos «inicializar una variable estructura» a condición de que ésta sea del tipo *extern* o *static*. Por ejemplo, si queremos declarar una variable «humano» como una estructura «editor» e inicializarla, tendremos que hacer lo siguiente:

static struct editor humano = {«juan»,«pez volador»,3490781}

El ejemplo anterior declara a «humano» como una estructura «editor».

Para referenciar a un miembro de una estructura se emplea una construcción de la forma:

«nombre de la estructura».miembro

En el ejemplo arriba expuesto, «humano.nombre» quedaría inicializado con «juan», «humano.direcc» con «pez volador» y «humano.teléfono» con «3490781». Como el lector habrá adivinado, el operador «.» conecta el «nombre de la estructura» con el «miembro». Veamos otro ejemplo sobre esto:

```
struct nómina
{long dni;
char empleado [20];
int código;
}
```

Si queremos definir una variable «agosto» como una estructura «nómina», solamente tenemos que colocar la siguiente línea:

struct nómina agosto;

En este momento la variable «agosto» tendría el mismo formato o estructura que la variable «nómina», y así podríamos manejar miembros «dni» con la estructura «agosto» de la forma «agosto.dni», que nos daría la información almacenada en el miembro «dni» de la estructura «agosto», etc.

En los modernos compiladores de C está permitido pasar estructuras como parámetros y devolver estructuras como valores de funciones, pero la comparación de estructuras está prohibida.



ARRAYS DE ESTRUCTURAS

Si comprendió bien cómo se declaraba un array, visto en el capítulo anterior, no tendrá ninguna dificultad para definir un array de estructuras, pues el proceso es completamente similar al de cualquier array.

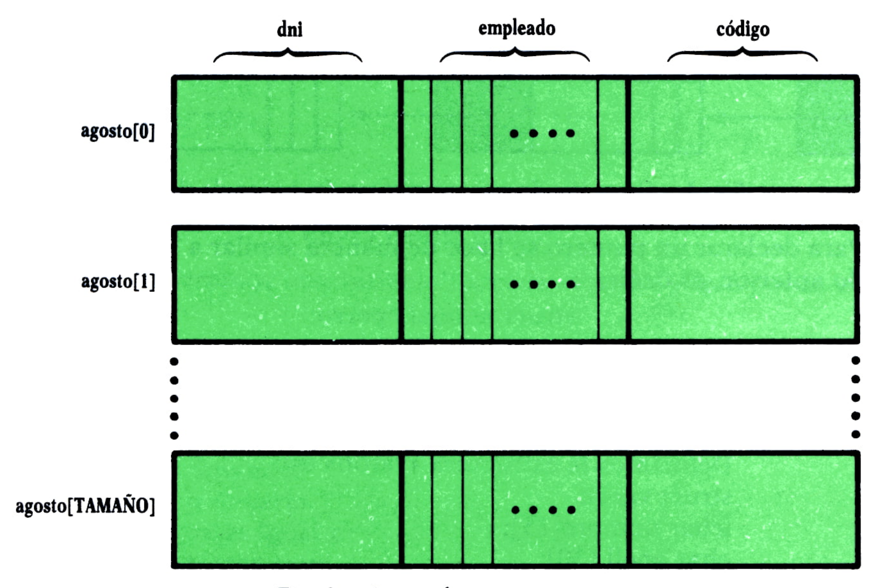


Fig. 2. Array de estructuras.

Por ejemplo si definimos:

struct nómina agosto[TAMAÑO];

estaremos declarando «agosto» como un array de «TAMAÑO» elementos, donde cada elemento del array es una estructura del tipo nómina. La identificación de los «miembros» de un array de estructuras se hace de forma análoga a como se hacía para una estructura sencilla, es decir, «nombre de la estrucutra» «.» «nombre del miembro».

Por ejemplo:

agosto[0].dni

es el documento nacional de identidad asociado con el primer elemento del array.



PUNTEROS A ESTRUCTURAS

En el capítulo dedicado a los arrays recordará qûe habíamos comentado que el uso de los punteros nos permitía tener una mayor flexibilidad en nuestro trabajo, al margen de otras ventajas. Pues bien, el uso de punteros a estructuras viene impuesto por la necesidad de tener una mayor flexibildad en el manejo de las estructuras además de permitirnos realizar el traspaso de estructuras como argumentos de funciones, ya que, como sabemos, una estructura no puede pasarse como argumento de una función.

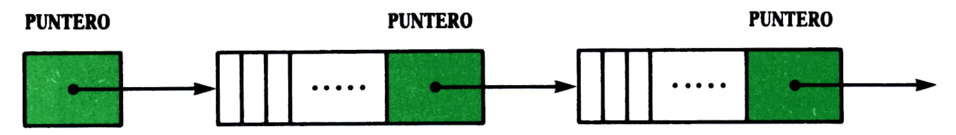


Fig. 3. Creación de una estructura dinámica de datos.

Para declarar un puntero se hace de manera similar a la vista en el capítulo anterior, es decir:

```
struct animal *perro;
```

Por supuesto, «animal» será una estructura definida y con sus miembros. En este caso «perro» será el nombre del puntero.

Veamos un ejemplo más completo:

```
Definamos la estructura «varón» así:
struct varón
{char nombre[15];
char apell [30];
}
```

las siguientes líneas nos mostrarán cómo trabaja un puntero a una estructura:

```
struct varón *puntero;
{printf («Nombre : %s \n",puntero -> nombre);
printf («Apellidos : %s \n",puntero -> apell);
}
```

Señalaremos que «puntero a estructura» seguido de «->» funciona exactamente de la misma forma que un «nombre de estructura» seguido del operador acceso «.» En otras palabras:

```
puntero -> miembro
es similar a
(*puntero).miembro
```

El operador «->» se compone de un guión «-» y el símbolo de mayor «>» del teclado de su ordenador.



UNIONES

El objetivo de las uniones en C es proporcionar una variable que permita almacenar diferentes tipos de datos en una única zona de memoria. Esto puede ser útil para la creación de arrays que contengan datos diferentes.

La sintaxis de una «unión» es similar a la de una estructura cambiando la palabra «struct» por «union». Veamos un ejemplo:

```
unión baraja
{int carta;
char palo;
}
```

Si ahora definimos una variable unión de tipo«baraja» tendremos:

```
unión baraja tomar;
```

Con esto declaramos una variable llamada «tomar» y que puede utilizarse así:

```
tomar.palo = 'c';
tomar.carta = 10;
```

También puede ser utilizado el operador -> con uniones de la forma:

```
p = &tomar;
y = p ->carta
lo que es igual a
y = tomar.carta
```

Las operaciones permitidas en las uniones son tener acceso a un miembro y tomar su dirección. Las uniones pueden aparecer dentro de estructuras y arrays.



CAMPOS

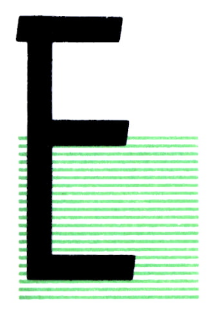
La principal utilidad de los «campos» reside en la posibilidad de manipular bits. El campo aparece dentro de una declaración de estructura de la siguiente forma:

```
struct
{unsigned auto: 1;
unsigned veri: 1;
}máscara;
```

La variable «máscara» contiene dos «campos» de 1 bit. Como habremos adivinado, los «campos» es la expresión constante que va a continuación del miembro de la estructura precedido por «:».

Para los tipos de miembros están permitidos «chart», «int», «short», «long» o «unsigned». El tamaño del campo (en bits) puede ser superior a uno. Un inconveniente de los campos estriba en que no pueden definirse arrays de campos ni definirse punteros a ellos.

FUNCIONES DE ENTRADA/SALIDA 8



EN este capítulo hablaremos de las funciones de la biblioteca estándar de E/S del C y cómo dichas funciones proporcionarán un sistema de E/S para los programas que se realicen en lenguaje C.



UTILIZACION DE LA BIBLIOTECA DEL C



Fig. 1. E/S.

Si deseamos utilizar una función de la biblioteca, en nuestro programa fuente, deberemos emplear la línea «#include <stdio.h>». Como observará, dicha línea ya ha aparecido en ejemplos de otros capítulos.

Dependiendo del sistema de que disponga, el acceso a la biblioteca será diferente. Debido a esto, cada caso particular podrá tener sus propias peculiaridades. Deberá observar si las funciones aquí definidas son aplicables a su sistema.

Como comentábamos en líneas anteriores, la línea

```
# include <stdio.h>
```

se emplea para que todo archivo fuente pueda utilizar las funciones de la biblioteca. El fichero <stdio.h> contiene macros y variables utilizadas por la biblioteca.



FUNCION PARA ABRIR FICHEROS

La función utilizada para la apertura de ficheros es «fopen()»:

FILE *fopen()

Veamos un poco más en detalle todo esto.

Antes de realizar cualquier tratamiento sobre un fichero hay que abrirlo mediante la función «fopen()» de la librería que C posee. La función «fopen()» está compuesta por tres parámetros:

— El primero es el «nombre» del fichero que queremos abrir, representado por un conjunto de caracteres.

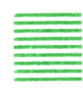
— El segundo el «modo» cadena de caracteres que indica la utilización que se va a hacer del fichero. Se permiten tres modos: «r» (lectura), «w» (escritura) y «a» (añadido).

— El tercero es un puntero al fichero:

```
FILE *puntarch
puntarch = fopen («ensayo», «r»);
```

Como vemos, «puntarch» es puntero al archivo «ensayo», por lo que en lo sucesivo el programa hará referencias al fichero por medio del puntero «puntarch».

La función «fopen()» devolverá un valor «NULL» en caso de que le sea imposible abrir el fichero.



MAS FUNCIONES DE E/S

Si tenemos que leer un carácter del dispositivo de entrada lo haremos a través de la función:

getc()

Por ejemplo:

ch = getc(puntarch);

nos indica que se ha de recoger un carácter del fichero al que apunta «puntarch».

Si deseamos enviar un carácter a un fichero apuntado por el puntero «puntarch» deberemos emplear «putc()» de la siguiente forma:

putc (ch,puntarch)

Como se vio en líneas anteriores, «puntarch» es un puntero de tipo FILE. La sintaxis general será:

putc(carácter, puntero al fichero)

Tanto «getc()» como «putc()» se utilizan sólo para ficheros de texto.
Para cerrar un fichero utilizaremos la función:

fclose()

Para cerrar el fichero apuntado por «puntarch» escribiremos:

fclose (puntarch);



FICHEROS ESTANDAR

Los tres ficheros estándar en C son:

«stdin» «stdout» «stderr»

Los tres ficheros son abiertos antes de «llamarse» a la función «main()». Estos tres archivos se denominan «entrada estándar», «salida estándar» y «salida estándar de errores».

Para estos tres archivos se declaran unos apuntadores a estructuras FILE, denominados «stdin», «stdout» y «stderr».

Para la entrada/salida de un solo carácter se utilizarán las dos funciones:

getchar() y putchar()

Estudiemos ambas por separado.

La función «getchar()» recoge un carácter del dispositivo de entrada (normalmente el teclado). Veamos un ejemplo:

```
#include <stdio.h>
main()
{char carácter;
carácter = getchat();
putchar (carácter);
}
```

El ejemplo anterior no tendrá dificultades para el lector, ya que lo único que hace este pequeño programa es recoger del teclado un carácter e imprimirlo a continuación.

Las definiciones de «getchar» y «putchar» están contenidas en el fichero <stdio.h>. Por esta razón, es necesario incluir dicho fichero al comienzo del programa.



SALIDAS Y ENTRADAS CON FORMATOS

Las funciones básicas para la salida y entradas con formatos son:

printf() y scanf()

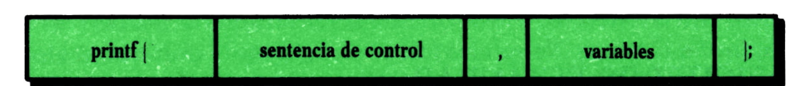


Fig. 2. Sintaxis de la función «printf()».

La función «printf()» nos permite convertir, dar formato e imprimir en el dispositivo de salida estándar una serie de argumentos definidos en la función, bajo el control del «parámetro de control».

La sintaxis utilizada será:

printf (argumento de control, argumento1,argumento2,....)

El parámetro de control establece dos tipos de salida: por una parte, los caracteres ordinarios que simplemente se copian a la salida estándar, y, por otra, las «especificaciones de conversión», que origina la impresión de los argumentos.

Cada una de las especificaciones de conversión comienzan con el carácter «%» y finalizan con un «carácter de conversión». Si entre el % y el carácter de conversión existe:

— Un signo «menos» (-) indicará que a la salida el argumento quedará ajustado a la izquierda.

— Una cadena de dígitos (número) indicará el tamaño mínimo del campo. Si el argumento convertido tiene menos caracteres que el tamaño del campo, se rellena por la izquierda hasta alcanzar el tamaño del campo.

— Un punto (.) separará el tamaño del campo de la precisión.

— Una cadena de dígitos (precisión) indicará el número máximo de caracteres que se imprimirán de la cadena, o el número de dígitos que se deben imprimir a la derecha del punto decimal de un valor «float» o «double».

— Una letra «l» indicará que el dato correspondiente es de tipo «long» en lugar de «int».

Los caracteres de conversión y lo que representan son:

— *d* Convierte el argumento entero en número decimal.
— *o* Convierte el argumento entero a octal.
— *x* El argumento se convierte a notación hexadecimal.
— *u* El argumento se convierte a decimal sin signo.

— *c* El argumento se interpreta como carácter.

— *s* El argumento es una cadena y se imprimirán todos los caracteres de la cadena hasta que se encuentre el carácter nulo o se alcance el número de caracteres especificado.

— *e* Convierte el argumento en coma flotante a notación exponencial con el formato [-]m.nnnnnnE[-]xx.

— *f* Convierte el argumento «float» o «double» a notación decimal de la forma [-]mmm.nnnnn.

— *g* Convierte el argumento en coma flotante a formato «%f» o «%e» dependiendo de cuál de los dos tenga la cadena de longitud más pequeña.

La función utilizada para la entrada con formato es:

scanf()

La sintaxis utilizada para esta función es:

scanf (argumento de control, argumento1, argumento2,...)

La función «scanf()» lee caracteres del dispositivo de entrada, los traduce de acuerdo al formato especificado y almacena los resultados en el resto de los argumentos.

Los argumentos especificados en la función «scanf()» deben ser punteros que indiquen dónde se debe almacenar el resultado de la conversión. Por tanto, cada uno de los argumentos deberá ir precedido del símbolo «&».

Los blancos, tabuladores o fines de línea contenidos en la cadena de control serán ignorados.

De igual manera, la cadena de control puede contener «especificaciones de conversión» formadas:

— El caracter %.
— El carácter * para indicar que la asignación queda en suspenso.
— Un número opcional que indica el tamaño máximo del campo.
— El carácter de conversión.

El carácter de conversión nos indica la interpretación que se hace del campo de entrada, y como se ha dicho anteriormente, los argumentos deben ser punteros. Como caracteres de conversión tenemos los siguientes:

— *d* Lectura de un número decimal a la entrada. El argumento debe ser un puntero a entero.

— *o* Lectura de un número octal a la entrada. El argumento debe ser un puntero a un entero.

— *x* Lectura de un número hexadecimal a la entrada. El argumento debe ser un puntero a un entero.

— *h* Lectura de un número entero «short» a la entrada. El argumento debe ser un puntero a un entero «short».

— *c* Lectura de un carácter a la entrada. El argumento debe ser un puntero a caracteres.

— *s* Lectura de una cadena de caracteres a la entrada. El argumento debe ser un puntero a un array de caracteres de un tamaño tal que pueda contener la cadena incluyendo un «\O» de finalización.

— *f* Lectura de un número flotante a la entrada.El argumento debe ser un puntero a «float».

Los caracteres de conversión «d» «o» y «x» pueden ir precedidos por la letra «l» (ele), que nos indicará que el argumento correspondiente es un puntero a «long» en lugar de a «int». Si la «l» (ele) precede a «e» o «f», el puntero es a «double» en lugar de a «float».

Veamos algunos ejemplos de todo esto:

```
int j;
float y;
char nombre[40];
scanf ("%d %f %s" ,&j,&y,nombre);
```

Si ahora introducimos por el dispositivo estándar de entrada la siguiente línea:

30 50.11E-1 Enrique

se producirán las siguientes asignaciones:

j = 30
y = 5.011
nombre = 'Enrique\O' (la cadena asignada a nombre y terminada por \O)

Veamos otro ejemplo:

```
int j;
float y;
char nombre[40];
scanf("%2d %f %*d %2s",&j,&y,nombre);
```

si ahora introducimos a la entrada la siguiente línea:

45678 9012 34b56

se producirán las siguientes asignaciones:

j = 45
y = 678.0
se salta 9012
la cadena '34'queda asignada a «nombre».

En sucesivas llamadas se comenzará la exploración en la letra «b».



FUNCIONES fprintf(), fscanf(), fgets() y fputs()

Las funciones «fscanf()» y «fprintf()» pueden ser utilizadas para realizar entradas y salidas con formato de archivo. Son similares a las ya estudiadas «scanf()» y «printf()», con la diferencia de que el primer argumento es el puntero al fichero del que se va a leer o escribir, y siendo la cadena de control el segundo argumento.

La función «fgets()» está orientada a líneas y utiliza tres argumentos. El primero es un puntero al lugar de destino de la línea que se va a leer El segundo argumento nos limita la longitud de la tira de caracteres que se lee. El tercer argumento es un puntero al fichero del que se está leyendo.

Veamos un ejemplo:

```
# include <stdio.h>
# define TAMAÑO 40
main ()
{FILE *fi
char *cadena [TAMAÑO]
fi = fopen («libro», «s»);
while (fgets(cadena,TAMAÑO,fi) !=NULL)
puts (cadena);
}
```

La función «fputs()» escribe una cadena de caracteres sobre el fichero especificado por el puntero. Por ejemplo:

```
fputs («Hola soy amigo tuyo», puntero);
```

enviará la cadena de caracteres «hola soy amigo tuyo» al fichero especificado por «puntero», que será un puntero de tipo FILE.

Mencionaremos en este apartado dos funciones más, incluidas en la librería estándar de C, que son similares en ciertos aspectos a «fgets()» y «fputs()». Nos estamos refiriendo a las funciones «gets()» y «puts()».

La función «gets()» utiliza un solo argumento y lee la siguiente línea de entrada de «stdin» (incluyendo el ñn) sobre el array de caracteres especificado. Es decir, se produce una captura de caracteres a través del dispositivo estándar de entrada (teclado), hasta encontrar un carácter «(\n)» (nueva línea). Veamos un ejemplo:

```
main()
{char ciudad[40]
printf("Cuál es tu ciudad?\n");
gets (ciudad);
printf ("Bonita ciudad, %s. \n",ciudad);
}
```

Podremos introducir un nombre de ciudad de hasta 39 caracteres de largo, ya que debemos reservar un espacio para '\0'.

La función «puts()» necesita un argumento que sea un puntero a una cadena (tira) de caracteres. Cuando «puts()» encuentra el carácter nulo final, lo sustituye por un carácter nueva línea y lo envía junto con la cadena de caracteres o tira. La función «puts()» escribe la cadena de caracteres sobre «stdout» (seguido de un carácter \n).



FUNCION DE ACCESO ALEATORIO

La funcion «fseek()» nos permite acceder directamente a un fichero. Algo parecido ocurre en Pascal con el comando «SEEK». En general, el acceso a ficheros se realiza de forma secuencial, pero la utilización de funciones como «fseek()» nos permiten un acceso más rápido a una determinada posición dentro de nuestro fichero. Deberemos tener presente que la función «fseek()» dispone de tres argumentos en su definición. El primero es un puntero a FILE (fichero objeto de búsqueda). El segundo argumento se denomina «offset» y nos indica la distancia a que debemos movernos desde el punto de comienzo, debiendo declararse de tipo «long». El tercer argumento identifica el punto de referencia para el «offset».

La sintaxis general sería:

fseek (fp,desplazamiento,modo)

Si modo = 0 «desplazamiento» se mide desde el principio del fichero.

Si modo = 1 «desplazamiento» se mide desde la posición actual del fichero.

Si modo = 2 «desplazamiento» se mide desde la posición final del fichero.

Así, fseek(fd,0L,0) se posiciona al final antes de escribir.

Tanto «modo» como el «puntero» (fd) son de tipo «int».

La función «fseek()» devuelve un valor «int».



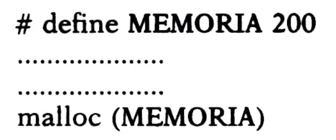
FUNCIONES DE ASIGNACION DE MEMORIA Y SALIDA

En ocasiones necesitaremos disponer de una cantidad de memoria suficiente que nos permita almacenar una tira de caracteres. Para realizar una asignación de memoria de forma automática disponemos de las funciones:

malloc() y calloc()

Como sabemos, en ciertos programas interactivos no sabemos «a priori» la cantidad de entradas que vamos a tener, por lo que lo que siempre hemos hecho con otros lenguajes de programación es hacer una reserva de memoria de forma que no tuviésemos problemas a la hora de ejecutar nuestro programa. En C, estas dos funciones nos permiten solicitar a lo largo del programa más memoria conforme se vaya necesitando. La importante ventaja de todo esto es obvio comentarla.

La función «malloc()» posee un argumento que representa el número de bytes que necesita en memoria. Por ejemplo:



```
# define MEMORIA 200
....................
....................
malloc (MEMORIA)
```

esta última línea reclama 200 bytes de memoria. La función devuelve un puntero «char» al comienzo del nuevo bloque de memoria.

La función «calloc()» se utiliza, igualmente, para hacer reserva de memoria y devuelve un puntero a «char», pero se diferencia de la anterior en que posee dos argumentos enteros y sin signo. El primer argumento informa a la función del número de células de memoria que se desean y el segundo es el tamaño de cada célula en bytes. Por ejemplo:

```
char *calloc();
long *reserva;
reserva = (long *) calloc (200,sizeof(long));
```

En este caso «long» utiliza cuatro bytes, y en consecuencia, la línea anterior nos da 200 unidades de cuatro bytes cada una, utilizándose en total 800 bytes.

Para salir de un programa y cerrar todos los ficheros abiertos se utiliza la función «exit()».

Esta función posee un único argumento, siendo éste un número de código de error. En la mayoría de los sistemas si este número es 0, nos dirá que la terminación se produce sin anomalías, pero si se dan valores diferentes, puede existir algún problema.

APENDICE A

SISTEMAS DE REPRESENTACION 



SISTEMA BINARIO

El sistema binario, de base 2, usa solamente dos dígitos o símbolos. Son el «0» y el «1». El valor posicional de los dígitos aumenta de dos en dos de derecha a izquierda.

10^5	10^4	10^3	10^2	10^1	10^0	
100.000	10.000	1.000	100	10	1	Decimal
2^5	2^4	2^3	2^2	2^1	2^0	
32	16	8	4	2	1	Binario

Fig. 1. Comparación del valor posicional.

En el siguiente cuadro podemos ver una comparación del sistema decimal y del sistema binario.

Decimal	Binario
1101	1101
$1 \times 10^3 + 1 \times 10^2 + 0 \times 10^1 + 1 \times 10^0$	$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
$1 \times 10 \times 10 \times 10 + 1 \times 10 \times 10 + 0 \times 10 + 1 \times 1$	$1 \times 2 \times 2 \times 2 + 1 \times 2 \times 2 + 0 \times 2 + 1 \times 1$
$1 \times 1000 + 1 \times 100 + 0 \times 10 + 1$	$1 \times 8 + 1 \times 4 + 0 \times 2 + 1$
$1000 + 100 + 0 + 1$	$8 + 4 + 0 + 1$
1101	13

1101 (binario) es igual a 13 (decimal)

Fig. 2. Comparación binario-decimal.

DECIMAL
1101
$1 \times 10^3 + 1 \times 10^2 + 0 \times 10^1 + 1 \times 10^0$

BINARIO
1101
$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

Expresando el número binario en forma de potencias de dos es el modo de encontrar el equivalente decimal.

A continuación damos una tabla de equivalencia.

Decimal	Binario	
	8421	← Valor posicional
0	0000	
1	0001	
2	0010	
3	0011	← Valor 2 + 1 = 3 en decimal
4	0100	
5	0101	← Valor 4 + 1 = 5 en decimal
6	0110	
7	0111	← Valor 4 + 2 + 1 = 7 en decimal
8	1000	
9	1001	
10	1010	
11	1011	
12	1100	
13	1101	
14	1110	
15	1111	

Fig. 3. Equivalencia decimal binaria.

Para «sumar números binarios» se hace de forma análoga a cualquier otro sistema de numeración.

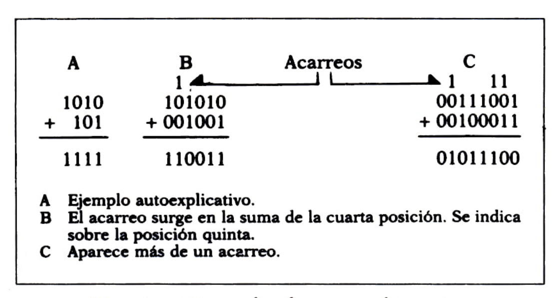


Fig. 4. Ejemplo de suma binaria.

Recordando que cuando se sumaba una unidad al dígito de orden más alto en decimal (9) se obtenía un cero (0) y quedaba una unidad de acarreo, en binario se actuará de un modo similar. Cuando se suma uno

más uno (1 + 1) que son los dígitos de mayor orden en binario, el resultado será cero (0) y se acarreará una unidad a la posición de orden inmediatamente superior.

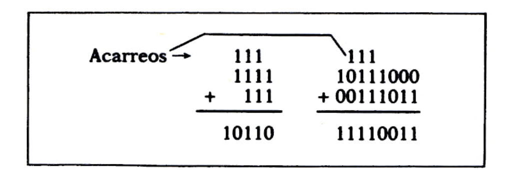


Fig. 5. Suma de varios 1 con acarreo.

Las reglas de adición en binario son:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 (y se acarrea 1)
1 + 1 + 1 = 1 (y se acarrea 1)

Para la «resta binaria» las reglas son muy simples:

0 - 0 = 0
1 - 1 = 0
1 - 0 = 1
0 - 1 = 1 con débito al bit de orden superior

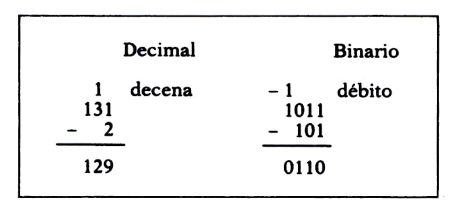


Fig. 6. Comparación resta decimal-binario.



SISTEMA HEXADECIMAL

Los números binarios se representan, en series de ceros y unos. A pesar de que este sistema es el que utiliza el ordenador de un nodo fácil, a todo el mundo se le hace más complicado trabajar con números binarios. En muchos grandes sistemas de ordenadores el sistema *hexadecimal* es el utilizado para representar números binarios simplificando su complejidad. De igual forma, el proceso de números en coma flotante se facilita mucho utilizando notación hexadecimal.

El sistema hexadecimal utiliza como base el número 16, por lo que este sistema tendrá 16 dígitos simbólicos. El numero de símbolos en un sistema de numeración es igual a la base del sistema. Binario o base 2 (dos símbolos 0 y 1), decimal o base 10 (de 0 a 9), hexadecimal o base 16 (de 0 a F).

Los 16 símbolos que se utilizan en hexadecimal son los 10 dígitos del decimal más las letras A, B, C, D, E y F. Se podrían utilizar otras cualquiera, pero por convenio se toman las seis primeras letras del alfabeto.

De este modo se representan los 16 dígitos con símbolos que ocupan una sola posición. Es decir si se usaran los símbolos 10, 11, 12, 13, 14 y 15 podría crearse confusión por estar compuestos por otros ya empleados en las diez primeras cifras y sería díficil distinguir si 10 eran 0 unidades de primer orden y 1 de segundo orden o 10 unidades de primer orden. Veamos la equivalencia entre dígitos decimales y hexadecimales.

Hexadecimal	**Decimal**
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

La suma hexadecimal sigue las mismas reglas que la binaria y la decimal, únicamente se utilizan símbolos distintos y la base es 16. Trabajar con símbolos alfanuméricos parece complicado al principio, pero con la práctica resultará, probablemente más sencillo. Ello requiere un cierto grado de mentalización. Por ejemplo mientras en decimal se dice 6 + 6 = 12, en hexadecimal se dice 3 + 8 = B (no 11).

Cuando la suma de dos dígitos excede de F(15) se acarreará una cifra al dígito de orden inmediatamente superior. De este modo en decimal 7 + 9 = 16, en hexadecimal 7 + 9 = 10, 8 + 9 = 11, A + 9 = 13 y así sucesivamente.

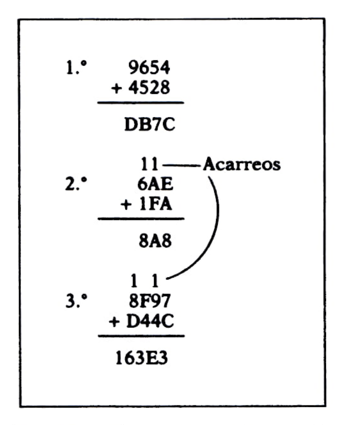


Fig. 7. Suma hexadecimal.

Siempre que se pase de F(15) se arrastrará una unidad a la cifra de orden inmediatamente superior. Veamos algunos ejemplos.

```
 9654
+4528
-----
 DB7C

  11   ← acarreos
  6AE
+ 1FA
-----
  8A8
```

+	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
1	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
2	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12
3	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13
4	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14
5	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15
6	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16
7	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17
8	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18
9	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19
A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A
B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B
C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20

Fig. 8. Tabla de suma hexadecimal.

Para la resta hexadecimal seguiremos reglas análogas a la resta decimal y binaria. Solamente se debe recordar el débito de la primera unidad cada vez que pase de 15 en una operación de sustracción. Por ejemplo

$$E - 5 = 9$$

	Decimal		Hexadecimal	
A				
	9	9		
	− 4	− 4	Mismo resultado	
	5	5		
B				
	11	11		
	− 4	− 4	Distinto resultado	
	7	D		
C				
	98	98 =	8 18	
	− 9	− 9 =	− 9	
	89		8 9	
				Modo erróneo
D				
	98	98 =	8 18	16 + 8 = 24
	− 9	− 9	− 9	24 − 9 = F (15)
	89		8 F	
				Modo correcto

A En decimal y en hexadecimal se obtienen los mismos resultados.

B En decimal se obtienen resultados distintos a los obtenidos en hexadecimal.

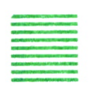
C Al 9 se le quita una unidad y se convierte en 8. Esta unidad se añade a las 8 unidades de orden inferior y tenemos un 18 hexadecimal (que en decimal es 24). Si a este 24 le restamos 9 nos da 15 en decimal, que es F en hexadecimal y no 9 como da el resultado.

D Esta es la solución correcta para el caso C. Esto puede obtenerse directamente, sin tener que convertir en decimal.

Fig. 9. Ejemplos.

+	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
1	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
2	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12
3	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13
4	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14
5	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15
6	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16
7	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17
8	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18
9	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19
A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A
B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B
C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20

Fig. 10. Tabla de suma y resta hexadecimal.



CAMBIOS DE BASE DE NUMERACION

Se puede convertir cualquier número a su equivalente en otra base cualquiera y no solamente a los sistemas mencionados anteriormente.

Si se encontrara un sistema capaz de representar 16 estados con símbolos diferentes dispondríamos de un medio para representar los octetos más cómodamente y con menos símbolos.

La conversión *binario-decimal* se realiza dividiendo primero los bits en grupos de 4. Luego, empezando por la derecha, se traduce cada grupo de 4 bits a sus correspondientes valores hexadecimales. Veamos un ejemplo de esta conversión.

1111100110110100011 = 0011/1110/0110/1101/0011

= 3 E 6 D 3

Para la conversión *hexadecimal-binaria* sustituimos cada dígito hexadecimal por su correspondiente grupo de 4 bits, rellenados con ceros hasta completar cuatro posiciones. Veamos un ejemplo de este tipo de conversión.

6C6F2E = 6 C 6 F 2 E

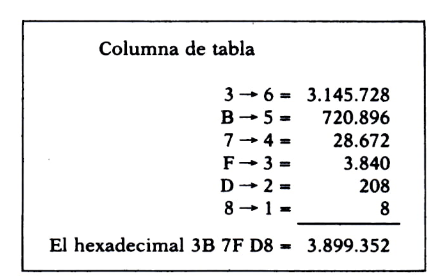
= 0110/1100/0110/1111/0010/1110

Para la conversión *hexadecimal-decimal* se requiere en principio cálculos matemáticos. Se puede utilizar la tabla de conversión que damos a continuación.

6		5		4		3		2		1
HEX = DEC		HEX = DEC		HEX = DEC		HEX = DEC		HEX = DEC		HEX = DEC
0	0	0	0	0	0	0	0	0	0	
1	1,048,576	1	65,536	1	4,096	1	256	1	16	
2	2,097,152	2	131,072	2	8,192	2	512	2	32	
3	3,145,728	3	196,608	3	12,288	3	768	3	48	
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	
0123		4567		0123		4567		0123		4567
BYTE				BYTE				BYTE		

Fig. 11. Tabla de conversión hexadecimal-decimal.

Supongamos que queremos convertir el número 3B7FD8 a decimal. A continuación se explica cómo utilizar la tabla para convertir este número a decimal.



Columna de tabla

3 → 6 =	3.145.728
B → 5 =	720.896
7 → 4 =	28.672
F → 3 =	3.840
D → 2 =	208
8 → 1 =	8
El hexadecimal 3B 7F D8 =	3.899.352

Fig. 12. Utilización de la tabla.

Para la conversión de *decimal-hexadecimal* se usa la misma tabla de la figura.

APENDICE B

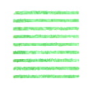
PALABRAS CLAVES EN C

Bucles: *for while do.*
Decisión: *if else switch case default.*
Tipos de datos: *chart int short long unsigned float double struct union typedef.*
Almacenamiento: *auto extern register static.*
Miscelánea: *return sizeof.*
Sólo para algunos sistemas: *asm endasm fortran enum.*
Todas las palabras anteriores son *palabras reservadas* y como tales no pueden utilizarse en un programa C como nombres de variables.

APENDICE C

TIPOS DE DATOS Y MODOS DE ALMACENAMIENTO





TIPOS DE DATOS

Tipos: int, long, short, unsigned, char, float, double.

Enteros con signo: Positivos o negativos.

int: Entero básico.

long o *long int:* Almacena un entero del tamaño, como mínimo del mayor *int.*

long será por regla general superior a short.

Enteros sin signos: Positivos o cero. Se utiliza para ellos la palabra clave *unsigned.*

Caracteres: Son símbolos y se almacenan en un byte de memoria.

char: palabra clave para este tipo.

Punto flotante: Valores positivos o negativos.

float: Punto flotante.

double o *long float:* Permite mayor número de cifras significativas.



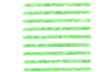
MODOS DE ALMACENAMIENTO

Las variables externas son las que se definen fuera de las funciones y tiene un alcance global. Las variables *automáticas* y globales son las que se definen dentro de la función.

Almacenamiento	**Duración**	**Alcance**
auto	temporal	local
registro	temporal	local
estático	permanente	local
externo	permanente	global

APENDICE D

ESTRUCTURAS DE CONTROL, RAMIFICACION Y SALTO



while

Bucle con una condición a la entrada.
Formato:
while (expresión)
sentencia;

Sentencia se repetirá hasta que la expresion deje de ser verdadera.

for

Esta sentencia utiliza tres expresiones de control: inicialización, test, actualización
Formato:
for (inicialización; test; actualización)
sentencia;

El bucle se repite hasta que *test* deje de ser cierto.

do while

El bucle se ejecuta hasta que expresión deje de ser cierta.
Formato:
do
sentencia
while (expresión);

if,else

Elección de opciones. Se produce una ramificación.

Formatos:

1. *if* (expresión)
   sentencia

2. *if* (expresión)
   sentencia A

   else
      sentencia B

3. *if* (expresión 1)
   sentencia A

   else if (expresión 2)
      sentencia B

   else
      sentencia C

En el formato 1 la sentencia se ejecuta si la «expresión es cierta».

En el formato 2 la sentencia A se ejecuta si la «expresión es cierta», caso de no serlo se ejecutará la sentencia B.

En el formato 3 si la «expresión 1 es cierta» se ejecuta la sentencia A, en caso contrario si la «expresión 2» es cierta se ejecuta la sentencia B, pero si ambas «expresiones son falsas», se ejecuta la sentencia C.

switch

Se produce un salto a la sentencia cuya etiqueta es igual al valor de la «expresión».

Formato:

```
switch (expresión)
{
case etiqueta1 : sentencia A
case etiqueta2 : sentencia B
case etiqueta3 : sentencia C
default : sentencia D
}
```

break

Cuando un programa llega a una sentencia *break* deja sin ejecución el resto del bucle o el switch que lo contiene.

continue

Puede ser utilizada con cualquier formato de bucle excepto con *switch*. El programa, una vez alcanzado un comando *continue*, deja de ejecutar las siguientes sentencias del bucle donde se encuentra situado.

goto

Se produce un salto incondicional. El salto es a la sentencia que posea la etiqueta indicada. No es aconsejable el uso de la sentencia *goto* en programación estructurada.

Formato:

```
goto etiqueta
----------------
----------------
etiqueta : sentencia
```

APENDICE E

OPERADORES Y CAMPOS 



OPERADORES LOGICOS

Trabajan cada «bit» de forma independiente.

˜: Negación de bits. Cambia los 0 a 1 y los 1 a 0.

&: AND: El resultado es 1 cuando los dos bits son 1.

|: OR: El resultado es 1 si alguno de los dos bits es 1.

ˆ: OR EXCLUSIVO: El bit resultante es 1 si alguno de los operandos contiene un 1 pero no cuando ambos lo contienen a la vez.



DESPLAZAMIENTO DE BITS

<<: Desplazamiento a la derecha: Este operador desplaza los bits del operando izquierdo a la izquierda el número de sitios indicado por el operador de su derecha, rellenando con ceros las posiciones vacantes.

>>: Desplazamiento a la derecha: Este operador desplaza los bits del operando situado a su izquierda hacia la derecha el número de posiciones que le marque el operando situado a su derecha, rellenando los lugares vacantes a la izquierda con ceros.



CAMPOS

Permiten una manipulación de bits muy eficiente. Deben definirse por medio de una «estructura» determinándose en ésta su anchura. Sólo se les puede asignar valores «0» y «1», ya que cada campo está compuesto de un bit.

APENDICE F

FUNCIONES DE LA LIBRERIA DEL C

La mayor parte de de las funciones de la librería de C requieren la utilización de algún fichero #include, que debería incluirse al principio del fichero a compilar.

Las funciones de librería son:

ABS: Aplicada sobre una variable nos devuelve su valor absoluto.

ATOF: Convierte una cadena de caracteres en su valor numérico expresado en doble precisión.

BSEARCH: Búsqueda binaria.

CEIL: Desprecia los decimales de una variable numérica con decimales.

CLOCK: Devuelve el tiempo de CPU que se ha utilizado.

CONV: Traslada caracteres.

CRYPT: Encriptación por clave.

CTERMID: El terminal conectado queda asociado a un nombre de fichero.

CTIME, LOCALTIME, GMTIME, ASCTIME, TIMEZONE: Convierte la representación interna de la fecha y la hora del sistema a un formato alfanumérico.

CTYPE: Ordenación del código ASCII según una tabla.

CURSES: Manejo de pantalla con optimización del cursor.

CUSERID: Devuelve el nombre del usuario conectado al terminal.

ECVT,FCVT,GCVT: Convierte un número en coma flotante a cadena de caracteres.

FABS: Devuelve el valor absoluto de un número.

FCLOSE,FFLUSH: Cierra un canal de entrada/salida o hace un vaciado de su buffer.

FDOPEN: Asocia un descriptor de fichero con un canal de entrada/salida.

FEOF: Cuando se alcanza el final de fichero devuelve un valor distinto de cero.

FERROR: Si se produce un error en lectura o escritura devuelve un valor distinto de cero.

FGETS: lee (n - 1) caracteres de un canal.

FILENO: Devuelve el número entero asociado al descriptor del fichero.

FLOOR: Para todo número con decimales devuelve el entero inmediato superior pero sin decimales.

FMOD: Devuelve la función resto (módulo).

FOPEN: Abre un fichero y lo asocia con un canal de entrada/salida.

FPRINTF: Escribe la salida, con formato, sobre un canal de entrada/salida.

FPUTC: Escribe un carácter sobre un canal de entrada/salida.

FREAD, FWRITE: Proporciona entrada/salida binaria.

FREOPEN: Realiza la sustitución de un canal de entrada/salida abierto por un fichero.

FSCANF: Realiza la lectura de datos de un canal de entrada/salida.

FSEEK: Origina que el puntero se posicione para la siguiente lectura/escritura en un canal de entrada/salida.

FTELL: Devuelve la posición actual del puntero en un canal de entrada/salida.

GETC, GETCHAR, FGETC, GETW: Toma un carácter o una palabra de un canal de entrada/salida.

GETENV: Realiza la búsqueda de un nombre en el entorno de ejecución del usuario.

GETGRENT, GETGRGID, GETGRNAM, SETGRENT, ENDGRENT: Actúa sobre las protecciones de grupos de usuarios de un fichero.

GETLOGIN: Da el nombre del usuario conectado al terminal.

GETOPT: Realiza una extracción de las letras de opciones de un vector.

GETPASS: Realiza una lectura de la clave de acceso (password) de un usuario.

GETPWENT, GETPWUID, GETPWNAM, SETPWENT, ENDPWENT: Para las entradas del fichero de usuario.

GETS: Realiza la lectura de un literal de un canal de entrada/salida.

LOGNAME: Devuelve el nombre de conexión (login) de un usuario.

LONGJMP: Recupera el stack del entorno de usuario guardado previamente.

MALLOC, FREE, REALLOC, CALLOC: Realizan asignación dinámica de memoria central.

MKTEMP: Crea un fichero único.

PERROR, ERRNO, SYS_ERRLIST, SYS_NERROR: Tratan los mensajes de error del sistema.

POPEN, PCLOSE: Inicia o termina la entrada/salida de un proceso a través de una tubería (pipe).

PRINTF: Imprime salida con formato.

PUTC: Coloca un carácter en un canal de entrada/salida.

PUTCHAR: Coloca un carácter en el fichero de salida estándar (stdout).

PUTS, FPUTS: Coloca un literal en un canal de entrada/salida.

PUTW: Coloca una palabra en un canal de entrada/salida.

RAND, SRAND: Genera números aleatorios.

REWIND: El puntero de acceso a un fichero es posicionado al comienzo del fichero.

SCANF: Realiza una lectura del canal de entrada estándar (stdin).

SETBUF: Asigna un buffer a un canal de entrada/salida.

SETJMP: Guarda el stack de ejecución de un proceso.

SLEEP: Durante un intervalo de tiempo especificado se suspende al ejecución de un proceso.

SPRINTF: Escribe un literal con salida formateada.

SSCANF: Realiza la lectura de un literal con formato.

STDIO: Entrada/salida estándar.

STRING: Realiza operaciones con literales.

STRNCAT: Añade una copia de «n» caracteres de un literal al final de otro.

STRCMP: Compara dos literales.

STRNCMP: Hace la comparación de al menos «n» caracteres de dos literales.

STRCPY: Copia un literal sobre otro.

STRNCPY: Copia exactamente «n» caracteres de un literal sobre otro.

STRLEN: Devuelve el número de caracteres distintos de nulo.

STRCHR: Devuelve un puntero a la primera ocurrencia del carácter «c» en el literal.

STRRCHR: Devuelve un puntero a la última ocurrencia del carácter «c» en el literal.

STRPBRK: Devuelve un puntero al primer carácter de un literal encontrado en otro literal.

STRSPN: Devuelve la longitud de un literal que coincide con otro literal.

STRTOK: Devuelve un puntero a la primera ocurrencia de un literal en el otro literal.

STRTOL, ATOL, ATOI: Convierten un literal en un entero.

SWAB: Realiza un intercambio de bytes.

SYSTEM: Proporciona un comando de la shell.

TERMCAP: Proporciona subrutinas para manejar el terminal, con especificación de los atributos de vídeo.

TEGENT: Coloca el nombre del terminal en un buffer.

TGETNUM: Devuelve el valor numérico del atributo "id" del terminal.

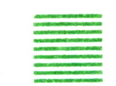
TGETSTR: Toma el valor literal del atributo «id» del terminal.

TGOTO: Devuelve un literal para efectuar el posicionamiento del cursor del terminal.

TMPFILE: Crea un fichero temporal.
TMPNAME, TEMPNAM: Asigna un nombre a un fichero temporal.
TTYNAME, ISATTY: Proporciona el nombre del terminal.
UNGETC: Devuelve un carácter a un canal de entrada/salida.

APENDICE G

AMPLIACIONES DEL LENGUAJE C



La versión 7 del sistema operativo UNIX recoge dos importantes extensiones al lenguaje de programación C. La primera de ellas es la utilización de una «estructura» (y no la dirección) como argumento de una función. La segunda nos permite la utilización de un nuevo formato de datos llamado «tipo de datos enumerados», formato de datos que los amantes del Pascal recordarán.

Como se ha visto a lo largo de este libro en C podemos pasar la dirección de la estructura a una función. Si «librería» es una estructura de tipo «libro» podemos llamar a una función de la siguiente forma :

```
veamos (&librería)
```

La función *veamos ( )* tendría un encabezamiento:

```
veamos(librero)
struct libro *librería;
```

Después de producirse la llamada a la función, librero apuntará a la estructura librería y la función utilizará librería en sus cálculos.

En el C extendido se puede utilizar el nombre de la estructura como argumento, lo que originará que se haga una copia de la estructura original en la función llamada.

Por ejemplo :

```
veamos (librería)
```

Con lo cual la función *veamos ( )* tendría un encabezamiento como:

```
veamos (librero)
struct libro librería;
```

Una vez llamada a la función, se origina una nueva variable de estructura de tipo *libro*. La nueva variable se denomina *librero*, y cada miembro de librero tendrá el mismo valor que el mienbro correspondiente de la estructura *librería*.

Esto supone la ventaja de eliminar los posibles efectos laterales no previstos dentro de la función, permitiendo a la función tener su copia de la estructura.

La palabra clave *enum* nos permite crear un nuevo tipo y especificar los valores que tiene. Por ejemplo:

```
enum arcoiris (rojo,violeta,azul,verde)
enum arcoiris colores;
```

La primera sentencia nos dice que se ha creado un nuevo tipo *:arcoiris*. Además tenemos el recorrido de dicho tipo. Este recorrido es el conjunto de valores que puede tomar una variable de tipo *arcoiris*: rojo, violeta, etc. Estos últimos serán constantes de tipo *arcoiris*.

La segunda sentencia declara *colores* como variable de tipo *arcoiris*. Se puede asignar a *colores* cualquiera de las constantes declaradas en *arcoiris*. Por ejemplo :

```
colores = violeta;
```

En principio los tipos *enum* son parecidos a los tipos ordinales definidos en Pascal, pero existen diferencias entre ambos.

Las operaciones que se pueden realizar con tipos *enum* son:

— Se puede asignar una constante *enum* a un variable *enum* del mismo tipo.
— Se pueden compara en test de igualdad o desigualdad.
— Se pueden usar operadores aritméticos en constantes *enum*.
— No se pueden utilizar operadores aritméticos en variables *enum*.
— No se pueden utilizar operadores de incremento y decremento.
— No se puede utilizar una constante *enum* como subíndice de un array.

APENDICE H

Tabla ASCII

DEX X_{10}	HEX X_{16}	OCT X_8	Binario $P X_x$	ASCII	Tecla*
0	00	00	0 000 0000	NUL	CTRL/1
1	01	01	1 000 0001	SOH	CTRL/A
2	02	02	1 000 0010	STX	CTRL/B
3	03	03	0 000 0011	ETX	CTRL/C
4	04	04	1 000 0100	EOT	CTRL/D
5	05	05	0 000 0101	ENQ	CTRL/E
6	06	06	0 000 0110	ACK	CTRL/F
7	07	07	1 000 0111	BEL	CTRL/G
8	08	10	1 000 1000	BS	CTRL/H, RETROCESO
9	09	11	0 000 1001	HT	CTRL/I, TAB
10	0A	12	0 000 1010	LF	CTRL/J, SALTO LINEA
11	0B	13	1 000 1011	VT	CTRL/K
12	0C	14	0 000 1100	FF	CTRL/L
13	0D	15	1 000 1101	CR	CTRL/M, RETURN
14	0E	16	1 000 1110	SO	CTRL/N
15	0F	17	0 000 1111	SI	CTRL/O
16	10	20	1 001 0000	DLE	CTRL/P
17	11	21	0 001 0001	DC1	CTRL/Q
18	12	22	0 001 0010	DC2	CTRL/R
19	13	23	1 001 0011	DC3	CTRL/S
20	14	24	0 001 0100	DC4	CTRL/T
21	15	25	1 001 0101	NAK	CTRL/U
22	16	26	1 001 0110	SYN	CTRL/V
23	17	27	0 001 0111	ETB	CTRL/W
24	18	30	0 001 1000	CAN	CTRL/X
25	19	31	1 001 1001	EM	CTRL/Y
26	1A	32	1 001 1010	SUB	CTRL/Z

DEX X_{10}	HEX X_{16}	OCT X_8	Binario P X_x	ASCII	Tecla*
27	IB	33	0 001 1011	ESC	ESC, ESCAPE
28	IC	34	1 001 1100	FS	CTRL<
29	ID	35	0 001 1101	GS	CTRL/
30	IE	36	0 001 1110	RS	CTRL/=
31	IF	37	1 001 1111	US	CTRL/-
32	20	40	1 010 0000	SP	ESPACIO
33	21	41	0 010 0001	!	!
34	22	42	0 010 0010	”	”
35	23	43	1 010 0011	#	#
36	24	44	0 010 0100	$	$
37	25	45	1 010 0101	%	%
38	26	46	1 010 0110	&	&
39	27	47	0 010 0111	’	’
40	28	50	0 010 1000	(	(
41	29	51	1 010 1001	)	)
42	2A	52	1 010 1010	*	*
43	2B	53	0 010 1011	+	+
44	2C	54	1 010 1100	,	,
45	2D	55	0 010 1101	-	-
46	2E	56	0 010 1110	.	.
47	2F	57	1 010 1111	/	/
48	30	60	0 011 0000	0	0
49	31	61	1 011 0001	1	1
50	32	62	1 011 0010	2	2
51	33	63	0 011 0011	3	3
52	34	64	1 011 0100	4	4
53	35	65	0 011 0101	5	5
54	36	66	0 011 0110	6	6
55	37	67	1 011 0111	7	7
56	38	70	1 011 1000	8	8
57	39	71	0 011 1001	9	9
58	3A	72	0 011 1010	:	:
59	3B	73	1 011 1011	;	;
60	3C	74	0 011 1100	<	<
61	3D	75	1 011 1101	=	=
62	3E	76	1 011 1110	>	>
63	3F	77	0 011 1111	?	?
64	40	100	1 100 0000		
65	41	101	0 100 0001	A	A
66	42	102	0 100 0010	B	B

DEX X_{10}	HEX X_{16}	OCT X_8	Binario P X_x	ASCII	Tecla*
67	43	103	1 100 0011	C	C
68	44	104	0 100 0100	D	D
69	45	105	1 100 0101	E	E
70	46	106	1 100 0110	F	F
71	47	107	0 100 0111	G	G
72	58	110	0 100 1000	H	H
73	49	111	1 100 1001	I	I
74	4A	112	1 100 1010	J	J
75	5B	113	0 100 1011	K	K
76	4C	114	1 100 1100	L	L
77	4D	115	0 100 1101	M	M
78	4E	116	0 100 1110	N	N
79	4F	117	1 100 1111	O	O
80	50	120	0 101 0000	P	P
81	51	121	1 101 0001	Q	Q
82	52	122	1 101 0010	R	R
83	53	123	0 101 0011	S	S
84	54	124	1 101 0100	T	T
85	55	125	0 101 0101	U	U
86	56	126	0 101 0110	V	V
87	57	127	1 101 0111	W	W
88	58	130	1 101 1000	X	X
89	59	131	0 101 1001	Y	Y
90	5A	132	0 101 1010	Z	Z
91	5B	133	1 101 1011	[	[
92	5C	134	0 101 1100	p	p
93	5D	135	1 101 1101	]	]
94	5E	136	1 101 1110	p	p
95	5F	137	0 101 1111	–	–
96	60	140	0 110 0000	p	p
97	61	141	1 110 0001	a	a
98	62	142	1 110 0010	b	b
99	63	143	0 110 0011	c	c
100	64	144	1 110 0100	d	d
101	65	145	0 110 0101	e	e
102	66	146	0 110 0110	f	f
103	67	147	1 110 0111	g	g
104	68	150	1 110 1000	h	h
105	69	151	0 110 1001	i	i
106	6A	152	0 110 1010	j	j

DEX X_{10}	HEX X_{16}	OCT X_8	Binario $P\ X_x$	ASCII	Tecla*
107	6B	153	1 110 1011	k	k
108	6C	154	0 110 1100	l	l
109	6D	155	1 110 1101	m	m
110	6E	156	1 110 1110	n	n
111	6F	157	0 110 1111	o	o
112	70	160	1 111 0000	p	p
113	71	161	0 111 0001	q	q
114	72	162	0 111 0010	r	r
115	73	163	1 111 0011	s	s
116	74	164	0 111 0100	t	t
117	75	165	1 111 0101	u	u
118	76	166	1 111 0110	v	v
119	77	167	0 111 0111	w	w
120	78	170	0 111 1000	x	x
121	79	171	1 111 1001	y	y
122	7A	172	1 111 1010	z	z
123	7B	173	0 111 1011	{	{
124	7C	174	1 111 1100	[	[
125	7D	175	0 111 1101	}	}
126	7E	176	0 111 1110	p	p
127	7F	177	1 111 1111	DEL	DEL

La amplia información del sistema operativo UNIX ha creado la necesidad de utilizar un lenguaje de programación que emplee los recursos de una forma eficaz.

Sin embargo, este lenguaje no sólo queda reducido al ámbito de los grandes sistemas que incorporan UNIX; también está adquiriendo una rápida popularidad entre los usuarios de los ordenadores personales, los amantes de la programación estructurada, los programadores que necesiten, para ciertos trabajos especiales, el tratamiento y manipulación de bits, los apasionados por los juegos, y un sinfín de características comentadas en este libro.

¿Qué lenguaje permite tener tantos adeptos? Desvelemos el misterio: el lenguaje de programación C.